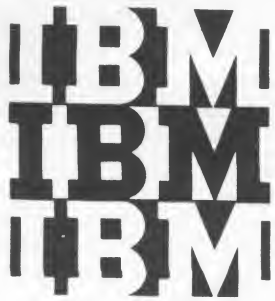


Form Y28-6612-1

Program Logic

IBM System/360 Operating System

Fixed-Task Supervisor

Program Number 360S-CI-505

This publication describes the fixed-task supervisor, which performs task management as a major part of the primary control program of IBM System/360 Operating System. In addition, this manual describes the initial program loader (IPL) and the nucleus initialization program (NIP).

Program Logic Manuals are intended for use by IBM customer engineers involved in program maintenance, and by system programmers involved in altering the program design. Program logic information is not necessary for program operation and use; therefore, distribution of this manual is limited to persons with program maintenance or modification responsibilities.

PREFACE

This manual describes the internal design of the fixed-task supervisor of IBM System/360 Operating System. Although this publication contains information concerning the supervisor in environments with a fixed number of tasks, this publication is issued only in support of single-task environments without protection. The external characteristics of this supervisor are described in the IBM Systems Reference Library.

Information in this document is directed to the customer engineer who maintains and services IBM System/360 Computing System and who is responsible for field maintenance and updating of IBM System/360 Operating System. This information may also be used by the programming systems maintenance programmer and the development programmer who will expand the system.

This publication may be used to locate those areas of the system to be analyzed or modified. The information is presented to enable the reader to quickly relate the task management functions to the program listings (coding) for those functions. The comments in the listings provide information for thorough analysis and understanding of the coding.

PREREQUISITE PUBLICATIONS

Knowledge of the information in the following publications is required for a full understanding of this manual.

IBM System/360: Principles of Operation, Form A22-6821

IBM System/360 Operating System: Concepts and Facilities, Form C28-6535

IBM System/360 Operating System: Control Program Services, Form C28-6541

IBM System/360 Operating System: Linkage Editor, Form C28-6538

IBM System/360 Operating System: System Programmer's Guide, Form C28-6550

IBM System/360 Operating System: System Generation, Form C28-6554

IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual, Form Z28-6605

Second Edition

This is a reprint of Z28-6612-0 incorporating changes released in the following Technical Newsletter:

Form Number	Pages Affected	Date
Z28-6612-0	65,66,67,68	March 31, 1966

Significant changes or additions to the specifications contained in this publication will be reported in subsequent revisions or Technical Newsletters.

With this edition, note that the prefix to the form number has been changed from Z28-6612 to Y28-6612. Distribution remains restricted.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.

A form for readers' comments appears at the back of this publication. It may be mailed directly to IBM. Address any additional comments concerning this publication to Programming Systems Publications, Department D58, PO Box 390, Poughkeepsie, N. Y. 12602

FIGURES

CHARTS

The fixed-task supervisor is a group of service routines that control the use of the central processing unit and main storage of IBM System/360. This supervision, called task management in the IBM System Reference Library, includes supervising the interfaces between processing programs and the primary control program. The primary control program is made up of the service routines for task management, data management, and job management. The fixed-task supervisor provides the following task management functions:

- Overlap of central processing unit operations with input/output channel activity.
- Servicing of all hardware interruptions.
- Handling of all supervisor calls (SVCs).
- Allocation of main storage for programs and data.
- Dynamic loading of programs not in main storage.
- Synchronous overlay supervision.
- Use of the hardware timer (optional).

The fixed-task supervisor is part of the primary control program, which is used to process batch jobs sequentially. The primary control program requires a main storage capacity of at least 32,768 bytes, and a minimum machine configuration that includes direct-access auxiliary storage.

MAIN STORAGE ORGANIZATION

In the single-task environment of the primary control program, main storage is divided into two areas: the fixed or system area, and the partition or processing program area. In expanded environments with a fixed number of tasks to be performed, main storage may be divided into the fixed area and two partitions, with one task using each partition, except when the higher-priority task (a teleprocessing task, for example) temporarily requires both partitions.

The fixed area is used for system routines that perform control functions during the execution of a processing program. The partition is used for a processing program and its data, control blocks, and tables.

The fixed area is divided into the nucleus and two transient areas. The nucleus contains the more frequently used SVC routines, the interruption handlers, and other routines and control information. The transient areas are two buffers into which less frequently used system routines are brought from the system residence. The first, called the SVC transient area, is 1024 bytes long and is used for SVC routines. The second, called the I/O supervisor transient area, is 400 bytes long and is used for the input/output supervisor's error handling routines.

PARTITION USAGE

A processing program is loaded into the lower section of the partition. Routines that the processing program has brought into main storage with a LOAD macro-instruction are placed in the upper section of the partition, the section with the numerically-greater main storage addresses. These routines, which may be system or user routines, remain in main storage for the duration of the job-step that loaded them, unless they are removed by using the DELETE macro-instruction.

When the processing program issues a LINK macro-instruction, the fixed-task supervisor loads the requested routine into main storage following the processing program. If this routine LINKs to another routine, the second routine follows the first in main storage. When one of these routines issues a RETURN macro-instruction, control returns to the program or routine that issued the LINK. For example, if routine A LINKs to routine B, routine B finishes and returns to A, and routine A then LINKs to routine C, the fixed-task supervisor overlays routine B with routine C. If routine A repeatedly LINKs to B, B stays in main storage. However, if A LINKs to C between uses of B, the supervisor overlays B in the interim period.

A routine that has been given control through a LINK macro-instruction and that has completed its operation and returned control to the routine that issued the LINK is termed inactive. A routine that is not inactive is termed active, implying that

the routine is currently executing, or has ceded control to another routine but will eventually resume control.

When a routine issues an XCTL macro-instruction, the main storage occupied by all inactive routines is freed. If the issuing routine was not brought into main storage with a LOAD macro-instruction, the storage occupied by the issuing routine is also freed. If the requested routine is not already in main storage, it is brought into the lower section of the partition.

INFORMATIONAL CONTROL BLOCKS

Processing programs that operate in a fixed-task environment do so as part of a task, a unit of work for the CPU. There is one task control block (TCB) for each partition, in which to record the addresses of pertinent information about the user's programs. This TCB is initialized by the nucleus initialization program (NIP) prior to any actual processing, and is used sequentially for each successive task performed by the system within this partition. (NIP is described in Appendix B.)

The TCB is 116 bytes long, with an additional 8 bytes at the end when necessary to support the timing option, and 32 bytes preceding the first byte when required as a floating point register save area. The format and contents of the TCB are given in the publication IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual.

There may be any number of programs (logically distinct sections of code) ready to be executed. Control passes from one such program to another by any of several means including a branch, LINK, XCTL, or ATTACH, or as the result of an interruption for which an asynchronous exit has been specified. Every transfer of control other than a direct branch is handled by the fixed-task supervisor.

Handling such transfers requires the maintenance of information allowing the supervisor to return control through the same sequence of programs but in reverse order. For example, if A links to B and B links to C, the supervisor must have the necessary information to return control to B when C completes operation and then to A when B completes operation. The request block (RB) is the repository for such information.

Request blocks are chained together to indicate the transfer of control. Each RB points to the RB for the program that will receive control when the program governed by the first RB has completed operation. The last element in the chain is the RB for the first program executed under the TCB. This RB points to the TCB instead of to another RB. In the preceding example, the RB for program C points to the RB for program B which points to the RB for program A, which points to the TCB. The TCB itself points to the RB most recently added to the queue, in this case the RB for program C.

Normally, one RB precedes the processing program and each requested routine. RBs are queued on the task control block. Those for active routines make up the active request block queue; those for inactive routines make up the inactive program list.

The first RB is placed on the active RB queue by NIP. An RB for job management is substituted for this first RB when NIP transfers control, via XCTL, to job management.

In addition to pointing to another RB or to the TCB, each RB contains the identification of the requested program, the entry point, the resume (interrupted) PSW, the size of the request block and the program, and the type of request block.

There are six types of request blocks:

- Program Request Block (PRB) -- used to control programs not previously loaded.
- Interruption Request Block (IRB)--used to control system or user asynchronous exit routines.
- System Interruption Request Block (SIRB) -- used to control I/O supervisor error routines.
- Supervisor Request Block (SVRB) -- used to control type 2 (resident), type 3 (non-resident, unimodular), and type 4 (non-resident, multimodular) SVC routines. Types 2, 3, and 4 SVCs may be enabled.
- Loaded Program Request Block (LPRB) -- used to control programs that are LOADed and are ATTACHed, LINKed, or XCTLed; also used to control sections of programs that are specified by the IDENTIFY macro-instruction and are ATTACHed.
- Loaded Request Block (LRB) -- shortened form of LPRB, used to control load modules that have the "only-loadable" attribute. (It is invalid to ATTACH, LINK, or XCTL to these load modules.)

The standard format for all request blocks and a description of their contents is given in the publication IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual.

REQUEST BLOCK QUEUEING

The TCB points to three RB queues: the active request block queue, the loaded program list, and the optional inactive program list. (See Figure 1.)

Active Request Block Queue

The active request block queue is a pushdown queue made up of PRBs, IRBs, SVRBs, LPRBs, and the SIRB. There is one RB for each program to be executed. The TCB, through the pointer named TCBRBP, points to the first (current) RB on the queue, and the last RB points back to the TCB. XRBLNK is the queueing field.

When there is an SIRB on the active request block queue, it is always the first on the queue (pointed to by the TCB). The routine associated with the SIRB is always the first executed.

Loaded Program List

The loaded program list contains LRBs and LPRBs in a two-way chain. Each loaded program is represented in this list. The TCB, through the pointer named TCBLLS, points to the first RB on the loaded program list. The RBs on the list are chained through the XRBSUC and XRBPRE fields. XRBPRE for the first RB in the queue points to the TCB. XRBSUC for the last RB on the list contains zero.

An LPRB may also appear on the active request block queue. In this case, it is maintained on both queues simultaneously through the two different sets of pointers.

Inactive Program List (Optional)

The inactive program list, a pushdown queue chained through the TCBJSE pointer in the TCB, contains PRBs removed from the active request block queue. The inactive list shows only programs still in main storage. The first program represented on the pushdown list is considered usable (if it is a reusable program). XRBLNK is the queueing field.

HOW THE FIXED-TASK SUPERVISOR IS ORGANIZED

The fixed-task supervisor is composed of the following major components, each of which is a functional grouping of supervisor service routines or subroutines: interruption supervision, task supervision, main storage supervision, contents supervision, program fetch, overlay supervision, and time supervision.

INTERRUPTION SUPERVISION

The interruption supervision service routines handle all interruptions on a first or introductory level. To do this they:

- Save information about the environment (machine status) at the time of the interruption so that the environment may be recreated later.
- Determine what action needs to be taken and set up the routines needed.
- Route control to the needed routines.
- Return to the interrupted environment.

TASK SUPERVISION

The task supervision service routines maintain control information. They maintain the current status of program and interruption request blocks, task control blocks, and event control blocks. The task supervision service routines are responsible for modifying task operations.

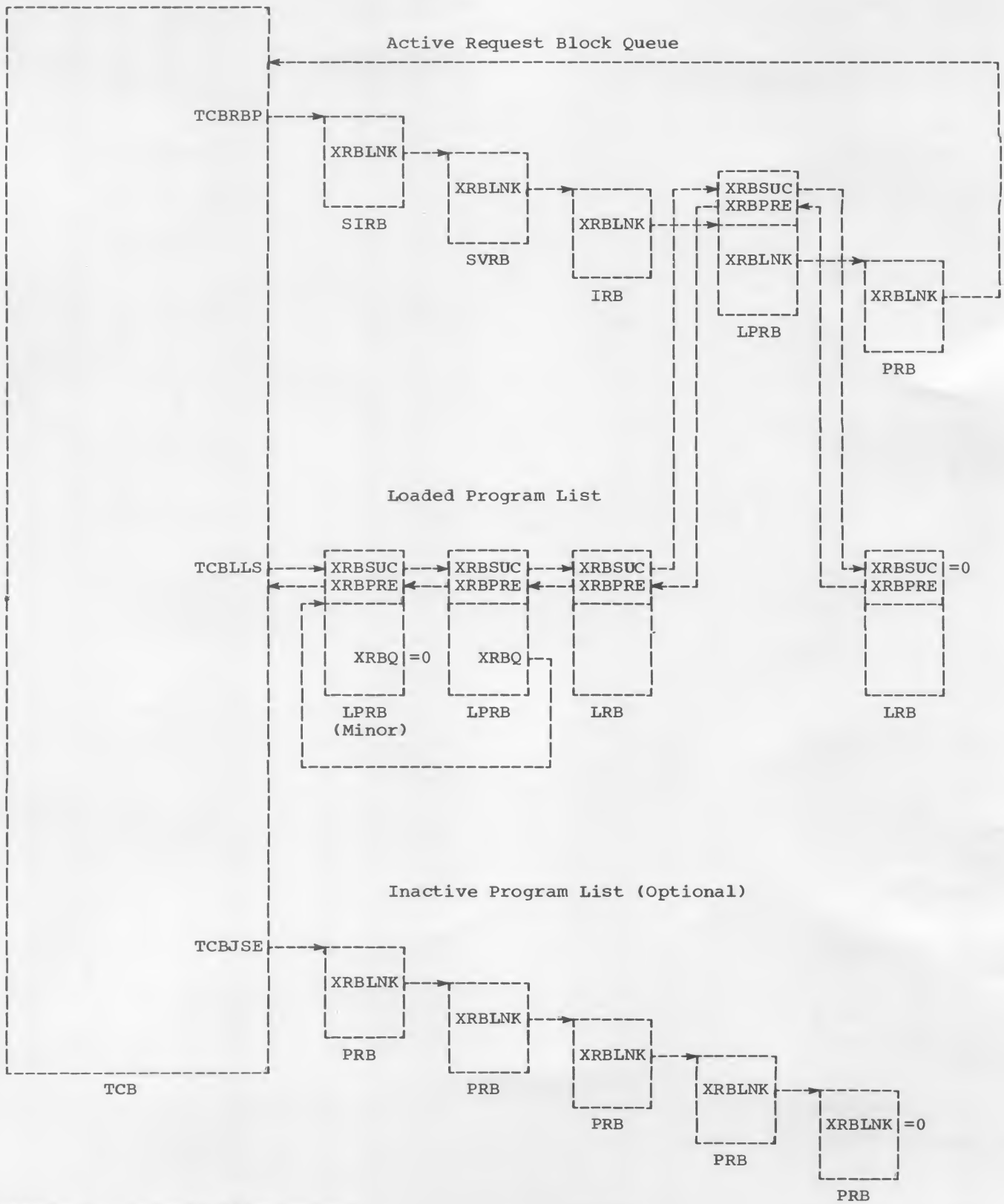


Figure 1. Request Block Queues

MAIN STORAGE SUPERVISION

The main storage supervision service routines establish the availability of main storage and dynamically allocate that storage to a task on request, within the partition associated with that task.

CONTENTS SUPERVISION

The contents supervision service routines maintain a record of the identity of all programs and routines together with their status and characteristics, within each partition. The contents supervision service routines initiate program fetch for the dynamic loading of programs, and maintain the active RB queue to represent requests for the use of programs.

PROGRAM FETCH

The program fetch service routine is a relocating loader which brings a program module processed by the linkage editor from secondary storage into a single area of storage.

OVERLAY SUPERVISION

The overlay supervision service routines monitor the flow of control between segments of a program operating in an overlay structure preestablished by the user through linkage editor. These routines ensure that all dependent program segments are brought into main storage by program fetch before the actual branch is executed.

TIME SUPERVISION

The time supervision service routines set and maintain a clock, and honor requests for time intervals and exact time.

FIXED-TASK SUPERVISOR CONTROL FLOW

As shown in Chart 00, flow in the fixed-task supervisor is in essence flow of interruption supervision, with alternate supplementary flow paths through other fixed-task supervisor components and other control program service routines -- those of data management, input/output supervision, job management, linkage editing, and test translation.

All interruptions in the central processing unit, in the channels, or in the devices attached to the channels, that affect control program processing, are placed before the interruption supervision service routines along with information identifying the cause of the interruption. These interruption handlers pass control to those parts of the control program that service individual interruptions.

When the interruption has been properly serviced, the interruption supervision service routines again receive control and return the central processing unit to the state in which it was operating before the interruption.

The CPU continues processing, but until another interruption occurs and brings it back into the supervisor state, it cannot execute privileged instructions -- it cannot execute channel instructions, storage protection instructions, or CPU-state-changing instructions other than SVC instructions.

CHAPTER 1: INTERRUPTION SUPERVISION SERVICE ROUTINES

Interruption supervision performs first level interruption handling: that is, the passing of control from processing program to control program and back again. To do this, the interruption supervision service routines must:

- Save the interrupted environment.
- Insulate interruption routines from each other.
- Exercise entry control to service routines required because of the interruption.
- *Return control* to the interrupted program at the completion of interruption handling.

In addition, interruption supervision provides through the SVC handlers all interface operations associated with the four types of supervisor call routines:

- Type 1 SVC routines. These are always resident and are executed disabled for their entire length. They usually effect return of control to the interrupted program without entering the dispatcher. A type 1 SVC may only call on other type 1 SVCs; this requesting of one type 1 SVC by another is called type 1 nesting. Examples of type 1 routines are GETMAIN, FREEMAIN, EXCP, WAIT, and EXIT.
- Type 2 SVC routines. These are also resident; but they are partially enabled, or they call on other than type 1 SVC routines. These routines are completely reenterable. Examples are LINK, LOAD, and XCTL.
- Type 3 SVC routines. These are like type 2 routines except that they are not resident. They are each brought into the 1024-byte SVC transient area. Examples are IDENTIFY, WTO, and LOCATE.
- Type 4 SVC routines. These are "multi-phase" type 3 routines. That is, they are too large to be brought into the transient area at one time and must be brought in in phases, each later phase overlaying an earlier one. Transfer of control from one phase to another is through XCTL. Examples are OPEN, CLOSE, and EOV.

To achieve a high response time for input/output interruptions, interruption supervision has a software-implemented disabling subroutine called the pseudo disable routine. This routine allows input/output interruptions to be processed without the requesting routine losing control -- the routine which was interrupted regains control as soon as the input/output supervisor has processed the interruption. Requesting routines include those system routines, such as the job management write-to-operator routine, that must operate enabled yet not lose control to another routine.

HOW INTERRUPTION SUPERVISION IS ORGANIZED

Interruption supervision is made up of the following service routines:

- SVC FLIH - The supervisor call first level interruption handler does the introductory work following an SVC interruption, and prepares for the execution of type 1 SVCs.
- SVC SLIH - The supervisor call second level interruption handler monitors the SVC transient area and prepares for the execution of types 2, 3, and 4 SVCs.
- Type 1 Exit - This routine is the exiting procedure for type 1 SVCs.
- EXIT - This SVC routine is the exiting procedure for types 2, 3, and 4 SVCs.
- Dispatcher - This routine passes control from routine to routine, whether system routine or processing program routine. Through two subroutines, the dispatcher sets up the mechanism to handle asynchronous exits and monitors the I/O supervisor transient area.
- I/O FLIH - The input/output first level interruption handler does the introductory work following an input/output interruption and the clean-up work after the input/output supervisor finishes second level handling.
- T/E FLIH - The timer/external first level interruption handler does the introductory work following any timer/external interruption and the clean-up work after the second level handling is completed.

- P FLIH - The program first level interruption handler monitors all program interruptions.

- PROLOG - This routine is used by P FLIH to set up input parameters to the ABTERM service routine of task supervision.

- MK FLIH - The machine check first level interruption handler routes all machine checks to system environment recording (SER) for second level handling, if SER is supported in the given environment. Otherwise, the machine is placed in a wait state.

- Validity Check - This routine is used as a common subroutine by other system routines, such as program fetch. The validity check routine prevents program interruptions caused by invalid addresses (those pointing beyond the boundaries of main storage) passed to the control program by a processing program. In installations that have selected the hardware protection option, this routine also checks for mismatch between the storage key of the addressed block and the protection key of the TCB associated with the processing program.

SVC CONTROL INFORMATION

The supervisor maintains SVC control information in the SVC table and the relocation table. These tables are in a module called IEASVC00, which is assembled at system generation time.

RELOCATION TABLE

The relocation table is used to relate the SVC code number with its corresponding entry in the SVC table. This relocation table consists of a number of 1 byte entries each of which is addressed through indexing based on the SVC code numbers. Each entry contains an index factor. If it is zero, then the associated SVC code is *invalid*. If non-zero then the factor gives the number of the entry in the SVC Table.

The relocation table is divided into two sections. The first section contains entries for IBM codes (that is, codes assigned to IBM-provided SVC routines) and there is one entry for each code number from 0 to but not including "High IBM code" in that order, whether or not that SVC code is in use in the system. The second contains entries for user codes, with one entry for each code number from 255 to but not including "Low User Code", in that order, whether or not the SVC is in use in the system.

The relocation table is variable in size with a maximum size of 256 bytes. Both the size and the contents of the table are determined at System Generation based upon the SVC routines included in the system. The relocation table format is shown in Figure 2.

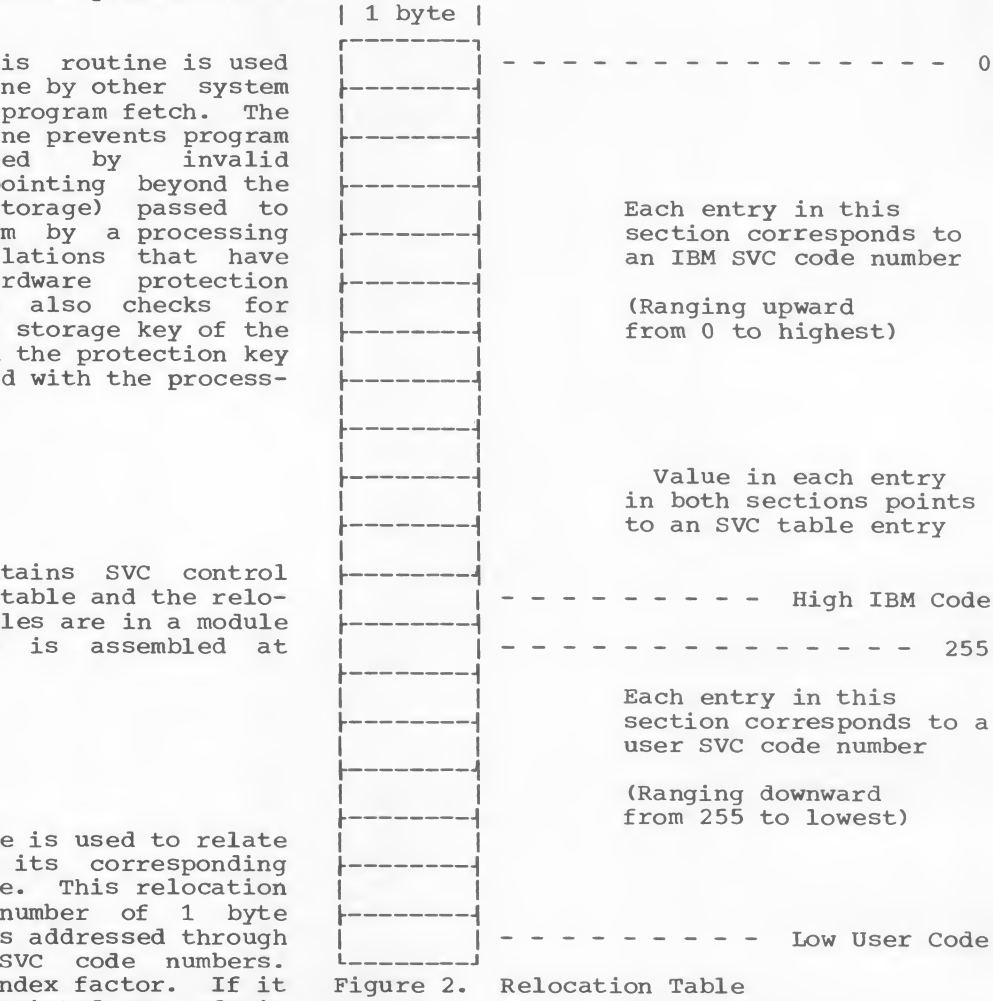


Figure 2. Relocation Table

SVC TABLE

The SVC table is divided into two sections. The first section consists of a 3-byte entry for each type 1 or type 2 SVC routine. The second section contains a 1-byte entry for each type 3 or type 4 SVC routine.

Each 3-byte entry contains a 24-bit main storage address with the three low-order bits defined as zero. This address is the address of an SVC routine. The three low-order bits of this address are used as a 3-bit field indicating the number of double-words required for an extended save area (ESA) in the RB. Each 1-byte entry contains the ESA information in the last three bits. If the three bits are zeros, a type 1 SVC is indicated. The SVC table and entry formats are shown in Figure 3.

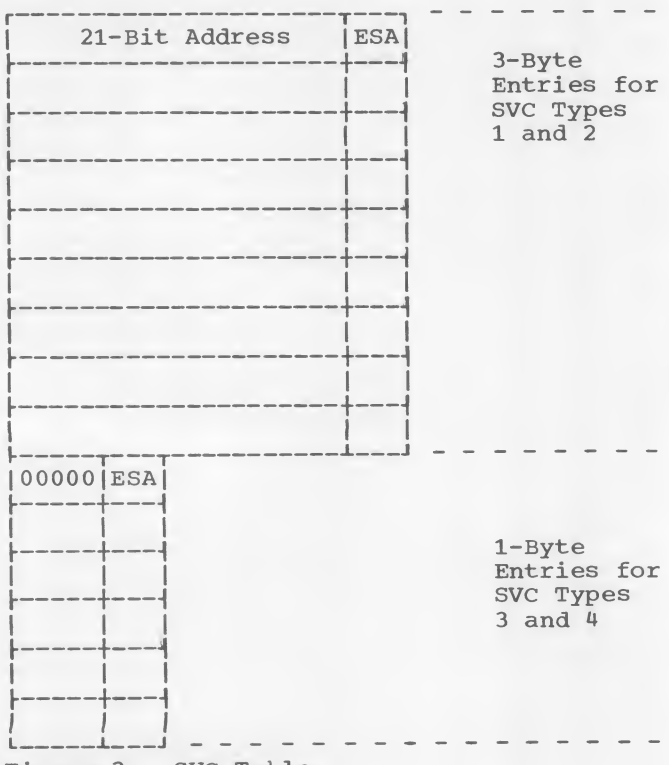
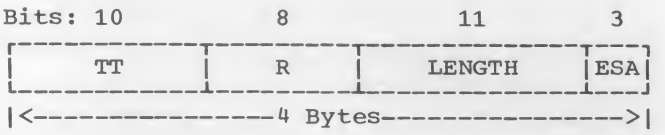


Figure 3. SVC Table

Optional Extension

The second section of the SVC table may be extended at system generation time so that each entry for a type 3 or 4 SVC routine is four bytes long instead of one byte. The 4-byte entry contains the track address (TT) of the transient SVC routine in the first field, the record number (R) on the track in the second field, the length of the first text record in the third field, and the size of the extended save area in the last field. The format for each entry is:



INTERRUPTION SUPERVISION CONTROL FLOW

The flow of control through interruption supervision, shown in chart 01, starts with an interruption. The five types of interruptions are SVC, input/output, timer/external, program, and machine check.

SVC INTERRUPTIONS

When an SVC interruption occurs, there are two paths to the requested SVC routine. These paths are described under SVC entry procedures. When the SVC routine completes, there are two possible paths of return. These paths are described under SVC exiting procedures. The dispatcher is discussed after the entry and exiting procedures, to show the flow back to the processing program.

SVC Entry Procedures

Entry to SVC routines is handled by the SVC FLIH and the SVC SLIH. The execution of any SVC instruction causes the hardware to give control to the SVC FLIH by loading a new PSW that is disabled for all maskable interruptions except machine check. The SVC instruction contains an 8-bit code which indicates to the SVC FLIH which service routine is required.

All registers are stored in the SVC save area, except when a type 1 SVC has issued an SVC. The SVC code is compared to the largest valid IBM-provided value plus one. If the code is equal to or larger than the maximum, the code is analyzed to determine whether the request is for a user-provided SVC routine. Abnormal termination of the task occurs if the requested SVC routine is not defined in the particular system configuration. If defined but unsupported (e.g., DETACH) it is treated as a no-operation (NOP).

Next, the SVC FLIH determines whether the requested SVC routine is listed in a resident SVC table. If listed, the address of the SVC routine is picked up from the table and used to enter the routine.

When the request is for other than a type 1 SVC routine, the FLIH branches to the SVC SLIH after moving the original register contents to the TCB. The SLIH creates SVRBs for types 2, 3, and 4 SVC routines. If the routine is a type 2 SVC, the SLIH passes control to the routine directly. If the routine is a type 3 or type 4, then control is passed only after it has been placed in the transient area via the FINCH routine (described in Chapter 4).

Specifically, the SVC SLIH first separates type 2 requests from types 3 and 4 so that the SLIH's SVRB creation and initialization subroutine can be executed immediately. For type 3 and 4 requests, the SVC SLIH initializes and, if necessary, fetches the required routines.

The SVRB creation and initialization subroutine stores the requestor's PSW in the current RB and then creates an SVRB for the called routine. The size of the SVRB is determined from the three low-order bits of the address in a full word. The address field of this full word is initialized by the SVC FLIH for type 2 requests and by the SLIH for type 3 and 4 requests, to contain in the three low order bits a value between 1 and 7. This value minus one is equal to the number of double-words of extended save area required by the called SVC routine.

The SVRB creation and initialization subroutine clears the three low-order bits of the address and saves the address in a register to preserve it across the GETMAIN which is issued for the SVRB. After getting the storage for the SVRB, the subroutine initializes it and queues it on the active RB queue.

If the SVC routine is a type 2, registers 0, 1, and 15 are restored from the save area of the SVRB, environmental registers are loaded, and the type 2 SVC routine is entered.

If the SVC is a type 3 or 4, the SLIH examines the SVC table, extracts information telling the size of the extended save area needed in the SVRB, and creates and initializes the SVRB.

If the current transient area occupant is not the requested routine, the requested routine must be loaded by FINCH, which is entered by a BALR. When the loading is completed, FINCH returns control to the SVC SLIH.

The separate phases of type 4 SVC routines bring each successive phase into the transient area by using XCTL until the phases are completed. The final phase issues an SVC EXIT instruction.

SVC Exiting Procedures

There are two exiting procedures for SVC routines -- type 1 exit and EXIT. Type 1 SVC routines with the exception of EXIT return to the type 1 exit for handling. Type 1 exit goes to the dispatcher for task switching or to the interrupted program -- either a processing program or a service routine. Types 2, 3, and 4 SVC routines return to the second procedure. EXIT dequeues the SVRB from the TCB's active RB queue and passes control to the dispatcher.

TYPE 1: Type 1 SVC routines branch directly to type 1 exit on completion. When given control, this exit procedure tests for SVC nesting. If nesting has occurred, the requestor of the exiting type 1 SVC routine is reentered by loading the SVC old PSW.

In the absence of nesting, registers are reloaded from the type 1 register save area of the SVC FLIH. The SVC old PSW is checked to determine if the requestor of the exiting type 1 SVC routine was disabled indicating that control is to be retained. If disabled, the requestor is reentered by loading the SVC old PSW. If the requestor was enabled, two full words, together called the TCB pointer or IEATCBP on the listing, are compared. If they are not equal, a task switch is indicated. Registers are saved in the task control block, the SVC old PSW is saved in the current RB on the active request block queue, and a branch is taken to the dispatcher. If a switch is not indicated, the requestor of the exiting type 1 SVC is reentered by loading the SVC old PSW.

EXIT: Types 2, 3, and 4 SVC routines, as well as asynchronous exit routines and routines entered by supervisor-assisted linkages, complete by using the EXIT routine directly or indirectly. Using EXIT directly means issuing an SVC EXIT instruction. Using EXIT indirectly means issuing a branch instruction with register 14 as an operand (or issuing a RETURN macro-instruction which expands to include a branch on register 14), where register 14 is preset by the supervisor to point to an SVC EXIT instruction in the nucleus.

EXIT determines the type of routine that is exiting, performs the necessary terminal procedures for the routine, and prepares for return to the routine in control prior to the exiting routine. In addition, EXIT determines if the routine to receive control is an SVC routine executed in the transient area. It is possible that the

sequence of events has caused the transient area to be overlaid since the SVC routine last had control. In this case, the transient area refresh subroutine of EXIT is entered to restore the SVC routine to the transient area.

EXIT passes control to either the dispatcher, a processing program, an asynchronous exit routine, or the task termination routine. The first and most common place is the dispatcher. The second, a processing program, is given control when the exit is from a program interruption routine. The third, an asynchronous exit routine, is given control when the exiting routine is an asynchronous exit routine and there are additional requests for the routine (RQEs) queued on the IRB under which it is operating. The fourth, the task termination routine, is given control when the returning program is the highest control level for a task.

When entered, EXIT resets the type 1 nesting switch because, although EXIT is entered as a type 1 SVC routine, it does not return through the normal type 1 exit. This is due to the peculiarity of being a transitional routine which passes control from one program to another.

After resetting the nesting switch, EXIT determines if the exiting routine is a program interruption routine. If it is, the address of a program interruption element (PIE) is loaded from the TCBPIE field of the TCB. The PIE contains the PSW and the contents of registers 14 through 2 that were in effect when the program interruption occurred, unless they were modified by the user's program interruption routine. The right half of the PSW saved in the PIE is moved to the SVC old PSW, registers 14 through 2 are loaded from the PIE register save area, and the SVC old PSW is loaded to return control to the processing program. Unless the user's program interruption routine modified the values in the PIE or in registers 3 through 13, the processing program regains control at the instruction following that which caused the program interruption.

If the exiting routine is not a program interruption routine, EXIT saves registers 10 through 1 in the register save area of the TCB and obtains the address of the RB for the exiting routine from the RB pointer field (TCBRBP) of the TCB and the address of the RB for the routine next to receive control from the XRBLNK field of the exiting RB. EXIT tests to see if the exiting RB is an IRB or the single SIRB in the system. (Both IRBs and the SIRB are discussed under Dispatcher and Exit Effector.) If it is either, EXIT determines if the RB has:

- Interruption queue elements (IQEs) with 4-byte link fields.
- IQEs with 2-byte link fields.
- No IQEs.

If the RB has interruption queue elements, the IQE at the top of the RB's XRBQ queue is removed. If the IQE has a 2-byte link field, the IQE is returned to the I/O supervisor to be placed on its list of available queue elements. (In the I/O supervisor program logic manual, IQEs with 2-byte link fields are called request elements.) Interruption queue elements with 4-byte link fields are not queued on any other queue and are effectively discarded when they are removed from the XRBQ unless the NEXAVL field of the IRB exists, in which case they are returned to this queue.

The RB is checked for more queue elements. If there are more, and if the new top IQE has a 2-byte link field, the address of the top IQE is loaded into registers 1 and 0. If the top queue element has a 4-byte link field, register 0 contains the address of the IQE, as before, but register 1 contains the data from the second 4-byte field of the queue element. In either case, the return address to be used by the asynchronous exit routine is loaded in register 14, and the entry point address of the asynchronous exit routine from the XRBEP field of the RB is loaded into register 15 before the routine is entered. The first word of the RB, potentially the register save area address, is loaded into register 13.

If there are no other IQEs queued on the RB, the saved registers are moved from the RB's register save area to the TCB's register save area. The exiting RB is dequeued from the active program queue of the task, and the routine to receive control is checked to see if it is in a wait state. If it is in a wait, the first word of the TCB pointer is set to zero, indicating that a task switch is necessary. If the RB is not waiting, the status bits in the RB for the routine to regain control are checked to see if the routine is a type 3 or 4 SVC. If it is, the name field in the RB (XRBNM) is compared to the name of the routine in the transient area. If the routine is not in the transient area, the transient area refresh subroutine is entered to bring it in. EXIT branches to the dispatcher.

Dispatcher

Loading a PSW to pass control to a routine associated with a request block is called dispatching. Dispatching is accomplished when EXIT, type 1 exit, I/O FLIH, or T/E FLIH branches to the dispatcher. The dispatcher gives control either to the routine last in control or to a different routine, or places the machine in a wait state. Dispatching a routine belonging to a task different than the task last in control, or placing the machine in the wait state, is called task switching.

Task switching occurs when the current routine in the current task can no longer be executed because:

- The current routine has issued a WAIT macro-instruction, setting the WAIT flag in the RB.
- A system routine has indicated that no routine in the current task can execute, by setting non-dispatchable bits in the TCB.
- A task of higher priority pre-empts the current task by becoming ready (in environments where the number of tasks is fixed but greater than one).

After receiving control, the dispatcher first schedules any requests for system asynchronous exit routines. Then it examines the first and second words of the TCB pointer. If the first word is not zero, it dispatches the task whose TCB is addressed. If the first word contains zero, the dispatcher searches for the first ready task on the queue of TCBs starting with the TCB addressed by the second word of the TCB pointer. (In a single-task environment, the TCB queue has only one TCB on it - the current TCB.) A ready task is one whose TCB has no non-dispatchable bits set and whose current RB is not waiting. In systems with the timer option (see Chapter 7), the dispatcher dequeues the timer element for a task time request before entering the wait state, and queues it again when leaving the wait state.

When dispatching a task, the dispatcher places the address of the task in both words of the TCB pointer, restores the registers, and loads the resume PSW. If there are no ready tasks, the machine wait state is indicated by turning on a bit in the resume PSW before loading it.

The dispatcher has a very important subroutine called the exit effector. The exit effector schedules the input/output supervisor's error routines using the I/O supervisor transient area and schedules requests to enter asynchronous exit routines by:

- Initializing an IRB or the SIRB.
- Placing the IRB or the SIRB on the active RB queue.
- Manipulating the saved registers to allow the dispatching of the asynchronous exit routine.

EXIT EFFECTOR: The exit effector consists of three parts. The first two parts are used by routines that require asynchronous exits. The third part is a dequeueing routine used by the dispatcher.

Part One: The first part of the exit effector is the CIRB service routine. This routine creates and initializes an IRB and, if specified, acquires additional storage within the partition for a register save area and a work area used for building interruption queue elements (IQEs). The address of the register save area is located in the three low-order bytes of the first word of the IRB. The format of the IRB is shown in Figure 4.

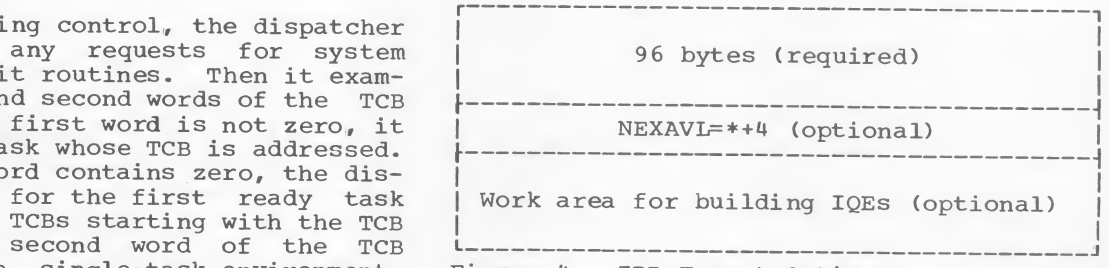


Figure 4. IRB Format Options

Part Two: The second part of the exit effector is used by a calling routine to schedule an asynchronous exit routine. Part two queues the IQE provided in register 1 as input, in FIFO order on either the 2-byte AEQ (asynchronous exit queue) or the 4-byte AEQ.

Part Three: The third, dequeueing part of the exit effector is entered by the dispatcher when the dispatcher finds that the AEQ points to an IQE. (Each time it is entered, the dispatcher checks for entries on the AEQ.) Part three dequeues the IQE from the AEQ, finds the IRB and TCB associated with the IQE, queues the IQE on the IRB and the IRB on the TCB's active RB queue. When two or more IQEs refer to the same IRB, they are queued in FIFO order.

Part three ensures that no IRB is scheduled for a task which has the SIRB on its active RB queue. The interruption queue element remains on the asynchronous exit *queue* to defer scheduling of the current IRB until the SIRB is inactive.

I/O SUPERVISOR ASYNCHRONOUS EXIT PROCESSING: The name of the error routine to receive control is generated using information in the UCB pointed to from the second half-word of the IQE. If the requested routine is in the I/O supervisor transient area, the routine is dispatched. Otherwise, FINCH (a routine described in Chapter 4) handles the interface with the data management BLDL routine and program fetch to load the error routine into the I/O supervisor transient area and ensures that the return address, entry point, and IQE address are in the registers and that the current error routine entry point is in the entry point slot of the SIRB.

EXITING FROM OTHER ASYNCHRONOUS EXIT ROUTINES: When the asynchronous exit routine for the first IQE is completed, EXIT is entered. The IQE is then dequeued from the IRB and is either returned to the I/O supervisor or queued on the NEXAVL field that immediately follows the IRB, or discarded.

If there are no additional IQEs queued on the IRB when an asynchronous exit routine returns, EXIT dequeues the IRB from the active RB queue. If there are additional IQEs queued on the IRB, the necessary initialization steps are executed and the IRB routine is reentered directly.

If the IRB and a work area were obtained by using part one of the exit effector, the work area is freed when the IRB is freed. If the IRB is to be reused, it is dequeued but is not freed.

INPUT/OUTPUT INTERRUPTIONS

Certain events, such as errors or completed actions in an input/output device or in the channel to which it is attached, cause the number of the device and a word of more detailed information about the status of the channel and the nature of the event to be placed in storage. The I/O FLIH is not concerned with the workings of the channel scheduler or with the inner details of input/output handling. It performs machine interruption supervision and insulates the input/output interruption from other types of interruptions. The I/O FLIH is given control by the input/output new PSW. The I/O FLIH is entered:

- Disabled for all maskable interruptions other than machine check.
- In supervisor state.

The first instruction of the I/O FLIH is a NOP/branch switch, set to a branch by the first input/output interruption, allowing input/output interruptions to be processed in groups. The first interruption of a group causes the I/O FLIH to execute some initialization instructions which block any further execution of this "first-time logic" for successive interruptions in a group. Registers two through nine are saved.

If the system is not pseudo disabled, the input/output old PSW is saved in the current RB. The wait bit in the input/output old PSW is set to zero (non-wait state), and registers ten through one are saved in the TCB's general register save area.

If the system is pseudo disabled, registers 10 through 1 are saved in the interruption supervision pseudo disable save area, and the input/output old PSW is saved.

I/O FLIH branches directly to that part of the input/output supervisor which handles interruptions. Upon return from the I/O supervisor, the NOP/branch switch is reset to no-operation. Registers 2 through 9 are restored.

The pseudo disable switch is tested. If off, the dispatcher is entered. If on, registers 10 through 1 are restored from the pseudo disable save area, and control returns to the interrupted routine by loading the input/output old PSW.

TIMER/EXTERNAL INTERRUPTIONS

Timer/external interruptions may come from the optional hardware timer at location 80, from the interrupt key on the console, and from six external units. The T/E FLIH in the fixed-task supervisor handles two kinds of timer/external interruptions: those caused by the optional hardware timer and those caused by the interrupt key on the console. The T/E FLIH passes control to time supervision for second level handling of timer interruptions and to job management's external interruption routine for second level handling of interrupt key interruptions.

When an interruption occurs, the hardware stores the current PSW in the timer/external old PSW location, indicates

the cause of the interruption in the interruption code field in the T/E old PSW, and loads the new PSW from the timer/external new PSW location. This gives control to the T/E FLIH.

The T/E FLIH saves registers 10 through 1 in the TCB, stores the timer/external old PSW in a standard original old PSW location (see program listing), and examines the interruption code in the timer/external old PSW to determine the interruption type.

When a supported interruption type is identified, the T/E FLIH branches to the appropriate second level handler. On completion of the second level handling, control is returned to the FLIH. A second, simultaneous interruption may have occurred, and the FLIH checks for this possibility, handling it in the same way as the first interruption.

After handling supported timer/external interruptions, the FLIH branches to the dispatcher. If non-supported timer/external interruptions occur, the T/E FLIH returns control immediately to the interrupted routine rather than to the dispatcher.

PROGRAM INTERRUPTIONS

If the program being executed attempts an improper action, a program interruption occurs and a code describing the attempt is stored in the program old PSW. Improper events causing program interruptions include addressing non-existent operation codes and attempting to execute privileged instructions. Users may specify fixed point overflow, decimal overflow, exponent underflow and significance as additional improper events requiring special handling.

If the user wishes to handle some or all program interruptions, he first issues a SPIE macro-instruction which generates a program interruption element (PIE) and inserts its address in the TCB. The program first level interruption handler (P FLIH) is given control by the hardware after any program interruption. The P FLIH checks the TCB for an address *of a PIE.* If no PIE address is present in the TCB, the interruption is unanticipated, and the P FLIH passes control to the PROLOG routine to initiate abnormal termination of the task.

If a PIE address is present in the TCB, the PIE is examined and the address of a program interruption control area (PICA) is extracted. The P FLIH tests the user's program interruption mask in the PICA to see if the user is handling the type of program interruption that has occurred. The type that has occurred is shown in the interruption code in the program interruption old PSW. If the user is handling the interruption, the P FLIH saves the old PSW and registers 14 through 2 in the PIE. Register 14 is loaded with a return address, register 1 with the address of the PIE, and register 15 with the address of the user's routine. The P FLIH places the address of the user's interruption routine, obtained from the PICA, into the old PSW, restores the work registers from the save area, and loads the modified old PSW to return to the user's program at the entry point of his program interruption handler.

The user may return to the main body of his program from his program interruption handling routine either by a direct branch or by issuing a RETURN macro-instruction. If the user returns to the main body of his program by a direct branch, he must reset the first-time-entry switch in the PIE.

If the program interruption type is not handled by the user, PROLOG is entered by a branch. This routine sets up the abnormal termination linkage and branches to ABTERM.

MACHINE CHECK INTERRUPTIONS

If the error detection equipment finds a machine error, information representing the internal state of the machine is placed in the diagnostic scan-out area of main storage. The hardware gives control to System Environment Recording or causes a wait state.

CHAPTER 2: TASK SUPERVISION SERVICE ROUTINES

The task supervision service routines maintain control information, cause tasks to be executed, and perform other task-related services. Task supervision service routines:

- Maintain task control blocks.
- Enter tasks into the wait state.
- Post completed events in the event control block.
- Maintain control levels indicated by request blocks.

HOW TASK SUPERVISION IS ORGANIZED

The task supervision service routines are functionally divided into two areas in the fixed-task supervisor: task modification and task termination.

TASK MODIFICATION

In the fixed-task supervisor, issuance of an ATTACH macro-instruction causes control to be given to a routine named by the issuer of the macro-instruction. The ATTACH service routine passes control to the requested routine and regains control when the attached program completes. ATTACH optionally posts an event control block to mark the completion, and, also optionally, passes control to a user-specified exit routine. If no special exit is specified, ATTACH returns control to the attaching routine.

Through the EXTRACT and SPIE service routines, task supervision allows the user to make better use of the system's controls. EXTRACT provides a processing program with information contained in specified fields of the task control block. SPIE allows the user to specify the address of an exit routine to be entered when specified program interruptions occur. The SPIE routine sets the program mask in the PSW as specified when a SPIE macro-instruction is given.

Through the WAIT and POST service routines, task supervision monitors the movement of the task between the ready and wait states. WAIT bars the continuation of the task until an event specified in the WAIT macro-instruction parameters has taken place and has been indicated by the execution of a POST macro-instruction. As an option, a WAIT routine to service multiple event completions may be chosen by the user. POST signals that the event represented by a specified event control block has occurred. This may result in a task being moved from a wait state to a ready state.

TASK TERMINATION

A task may be terminated by itself or by the system. Task supervision completes a task's execution through ABTERM and ABEND service routines. The ABTERM service routine schedules the ABEND routine, which terminates the task. The ABDUMP service routine is used when a full storage dump is required.

TASK SUPERVISION CONTROL FLOW

As shown in Chart 02, flow of task supervision is the flow of the individual modular service routines. Each receives control from interruption supervision and returns control to its particular exiting procedure. The one exception is the Abterm routine, which is branched to by any service routine, and returns to that routine by a branch.

ATTACH

The ATTACH service routine searches for the RB of the requested routine in the inactive program list and in the loaded program list. If the requested routine is not in the partition, ATTACH uses FINCH to bring it in. ATTACH places a request block on the RB queue for the attached routine. Control is given to the attached routine by loading a PSW with an LPSW. The request block queue is ordered as follows:

- RB for the attached routine.
- SVRB for the ATTACH routine.
- RB for the attaching routine.

When the attached routine completes, the ATTACH routine is dispatched and optionally posts the event control block. If the attaching routine specified an exit routine

in the ETXR parameter of the ATTACH macro-instruction, ATTACH places a request block on the active RB queue for the exit routine. When the ATTACH routine completes, the exit routine is dispatched, if this option was specified. When the exit routine completes, the attaching routine is dispatched.

EXTRACT

The EXTRACT service routine is entered from interruption supervision when the EXTRACT macro-instruction is issued. Upon entry, EXTRACT zeroes all fields in the list area specified by the user, except for the task input/output table (TIOT) address field. If the macro-instruction's parameters specified TIOT or ALL, the address in the TCB of the TIOT is inserted into its respective field in the user's list. *EXTRACT* issues an SVC EXIT instruction on completion.

SPIE

The SPIE service routine is used to set up indications that the user has requested program interruption control. SPIE is entered by the SVC SLIH when a SPIE macro-instruction is given. Thirty-two bytes of main storage space for a program interruption element (PIE) is obtained, and the address of the PIE is saved in the TCB. In creating the PIE (Figure 5), the SPIE routine places in the first four bytes the address of the program interruption control area (PICA) specified by the processing program in the SPIE macro-instruction. The SPIE routine sets aside the second eight bytes as a program interruption old PSW save area, and the next 20 bytes as a 5-register save area.

A program mask whose contents is determined by the interruptions selected is stored into the caller's resume PSW. SPIE executes an SVC EXIT instruction on completion.

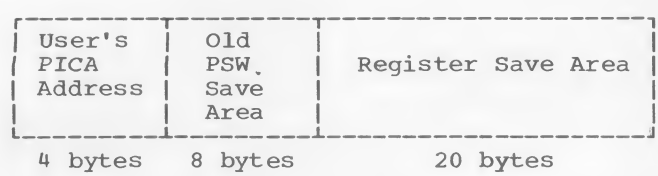


Figure 5. Program Interruption Element (PIE) Format

WAIT -- SINGLE EVENT

When WAIT is entered by the SVC interruption handler, the wait count passed as a parameter of the WAIT macro-instruction is tested for zero. If it is zero, the routine returns immediately by branching to the type 1 SVC exit. If it is non-zero, then the resume PSW of the caller is enabled for input/output and external interruptions. The wait and complete bits are tested in the ECB whose address was passed by the macro-instruction. When the complete bit is on, indicating that this event is already completed, WAIT branches to the type 1 exit. If the wait bit is on, indicating this event is already being waited for, WAIT terminates the task by branching to ABTERM. (Checking the wait bit is performed only if the validity check option is selected during system generation.) If neither bit is on, the wait bit is turned on and the address of the current RB is placed in the ECB. A wait count of one is placed in the current RB, and the first word of the TCB pointer, IEATCBP, is zeroed as a signal to the dispatcher that the task is waiting. WAIT returns by branching to the type 1 exit in interruption supervision.

WAIT -- MULTIPLE EVENT

The WAIT service routine is entered by the SVC FLIH as a result of a WAIT macro-instruction. Upon entry to the WAIT routine, the wait count passed as a parameter is tested for zero. If it is zero, the routine returns immediately by branching to the type 1 SVC exit. If the wait count is non-zero, the resume PSW of the caller is enabled for input/output and external interruptions. The wait count is saved and a loop initialized to address the ECBs addressed by the macro-instruction parameter list. An ECB counter is incremented as each ECB is addressed.

As in single-event WAIT, on an optional basis, the wait bit in the first ECB is tested. If it is on, indicating that this ECB is already being waited on, the next ECB is addressed. If the wait bit is off, the completion bit is tested. If the completion bit is off, indicating that a POST has not yet occurred, the wait bit is turned on and the address of the current RB is placed in the ECB. If this event has already completed -- if the completion bit is on -- the wait count is decremented and tested for zero. If the count is not zero, a test is made to see if this address is the last element in the parameter ECB list.

If it is not the last element, the cycle is repeated. If it is the last element, the loop is exited. If the wait count becomes zero, *all the wait* bits in the ECBs are turned off and the WAIT routine exits to the type 1 exit, without putting the current RB into a wait state since its count has already been satisfied.

When all ECBs have been addressed and the wait count has not become zero, the total number of ECBs specified is compared to the original wait count. If the number of ECBs specified is less than the count, the count cannot be satisfied; the task is abnormally terminated by scheduling ABEND through a branch to ABTERM.

If the wait count is less than the number of ECBs, a bit is turned on in the RB to indicate to POST that a multiple-event WAIT has been issued where the number of ECBs is greater than the wait count. If the wait count is less than or equal to the number of ECBs, WAIT inserts the wait count into the current RB and sets the first word of the TCB pointer to zero as a signal that the task is waiting. The WAIT service routine returns by branching to the type 1 exit routine of interruption supervision.

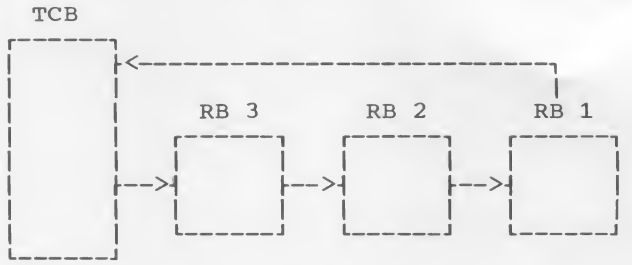
POST

The POST service routine is entered by the SVC FLIH after a POST macro-instruction is issued, but an alternate entry is provided so that system routines can branch directly to POST. Upon entry, POST tests the completion bit of the ECB whose address was passed as an input parameter. If it is on, indicating that the ECB has already been posted, the POST routine returns by branching to the type 1 exit or to the system routine which entered POST.

If the completion bit is off, the wait bit is tested to see if this event is being waited on. If the bit is off, the completion code is placed in the ECB and the completion bit is turned on. If the wait bit is on, the RB wait count is decremented, the completion code is placed in the ECB, the completion bit is turned on, and the wait bit is turned off. POST returns by branching either to type 1 exit or to the system routine which branched to POST.

In systems with a multiple event WAIT, POST performs further operations. When the wait count in the RB is decremented to zero, POST tests a bit in the waiting RB to see if the number of ECBs specified in the associated WAIT was greater than the wait count specified.

If the number of ECBs was greater, then POST turns off the wait bits in all ECBs in the ECB list specified which have not yet been posted, to indicate that no one is waiting for these events to be completed and to prevent an erroneous POST. The address of the ECB list is located in a register save area belonging to an RB or to the TCB. POST finds the addresses by determining which RB is waiting. If RB 3 in the following diagram is waiting, the address of the ECB list is in the register 1 field of the TCB register save area. If RB 2 is waiting, the list address is in the same field of the register save area of RB 3. If RB 1 is waiting, the address is in the register save area of RB2.



If the number of events waited on equals the number of events specified, the wait bits are turned off by POST as the events complete. After turning off the wait bits, POST places the completion code in the ECB, and returns.

RESIDENT ABNORMAL TERMINATION ROUTINE (ABTERM)

Certain system routines branch to the ABTERM service routine to schedule the abnormal termination of a task. ABTERM returns to the system routine by branching to the address passed to ABTERM in register 14.

When entered by a type 1 SVC routine, ABTERM saves the right half of the SVC old PSW and replaces the right half with the address of an SVC ABEND instruction. The task completion code is stored in the TCBCMP field provided in the TCB. After turning off the type 1 nesting switch in the SVC FLIH, ABTERM loads registers 0 and 1 from the type 1 SVC save area, restores registers and branches on register 14 as set by the SVC routine which branched to it.

When entered by any other system routine, ABTERM locates the current RB on the RB queue of the TCB, saves the wait count from the RB, replacing it with a zero wait count, and saves the right half of the

resume PSW from this RB. The task completion code is stored in the TCBCMP field in the TCB. ABTERM replaces the right half of the resume PSW in the RB with the address of an SVC ABEND instruction, restores the registers and branches on register 14 as set by the system routine which branched to it.

ABEND

The ABEND service routine is a type 4 SVC routine that is used for both normal and abnormal termination of tasks. The basic function of ABEND is to terminate all internal activities of the current task and give control via XCTL to the GO module of job management to continue processing.

Normal End

When ABEND is entered for a normal termination, it checks if all data sets have been closed. If any data sets are still open, ABEND calls the data management CLOSE routines. The task completion code is stored in the TCBCMP field of the TCB, and all main storage within the task's partition is designated as a free area. ABEND then XCTLs to job management to initiate either the next step of this job or the first step of a new job.

Abnormal End

When ABEND is entered for an abnormal termination, it checks if ABTERM was entered and if it was, ABEND restores the PSW and wait count to the *RB that called* ABEND. If ABTERM was not entered, ABEND stores the completion code in the TCBCMP field of the TCB. ABEND purges all input/output operations initiated for this task using the HALT I/O option. It performs validity checking of the various system queues -- such as main storage supervision queues, contents supervision queues, and data management queues -- to prevent ABEND from being requested while ABEND is in progress. ABEND removes the SIRB from the active RB queue.

ABEND determines the amount of main storage it will need and acquires the storage either by using GETMAIN or by overlaying reentrant code at the beginning of the partition.

ABEND checks if the abnormally terminating routine has requested a dump. If a dump was requested, ABEND searches the TIOT for a SYSABEND ddname. If this entry is not located, ABEND assembles pertinent information and packs it in main storage for eventual printing by the job management routines. This information is referred to as an indicative dump. If the SYSABEND entry was located, ABEND opens a DCB and calls a type 4 SVC routine named ABDUMP. ABDUMP assembles a full hex-formatted dump of the TCB, PSW, RBs, save areas, and all of main storage.

Upon completion of either the indicative dump or ABDUMP, or if no dump was taken, ABEND attempts to CLOSE all data sets by calling the data management CLOSE routines. As in normal termination, all main storage within the partition is designated as a free area. ABEND XCTLs to job management to print the indicative dump if provided and to initiate the next task.

CHAPTER 3: MAIN STORAGE SUPERVISION SERVICE ROUTINES

The main storage supervision service routines establish the availability of main storage space and dynamically assign space for program loading and work areas. Within each partition, the main storage supervision service routines:

- Allocate main storage space dynamically.
- Release main storage space dynamically on request.
- Maintain a record of all free areas of main storage.

HOW MAIN STORAGE SUPERVISION IS ORGANIZED

Main storage supervision is permanently resident within the nucleus, is not reenterable, and is disabled for all maskable interruptions except machine check. It is made up of the GETMAIN and FREEMAIN service routines.

The GETMAIN service routine allocates storage to a task according to its needs, when a GETMAIN macro-instruction is issued.

The FREEMAIN service routine releases storage space on request, when a FREEMAIN macro-instruction is issued.

MAIN STORAGE SUPERVISION CONTROL FLOW

As shown in Chart 03, the flow of main storage supervision is the flow of the service routines. The GETMAIN and FREEMAIN routines receive control from the SVC FLIH, and give up control through type 1 exit. Register-type GETMAIN and FREEMAIN requests have a separate entry point. An exception occurs when an error condition is encountered. In this case, control passes to ABTERM by means of a branch.

In the introduction to this manual, main storage was described as being separated into at least two areas, the fixed area and the partition (see Figure 6). The partition is the area subject to the fixed-task supervisor's storage allocation algorithm. This algorithm allocates space in the upper (higher address) portion of the partition to LOADed routines and data areas requested by the user, and space in the lower (lower address) portion to the processing program itself and to routines it has called through LINK, XCTL, and ATTACH.

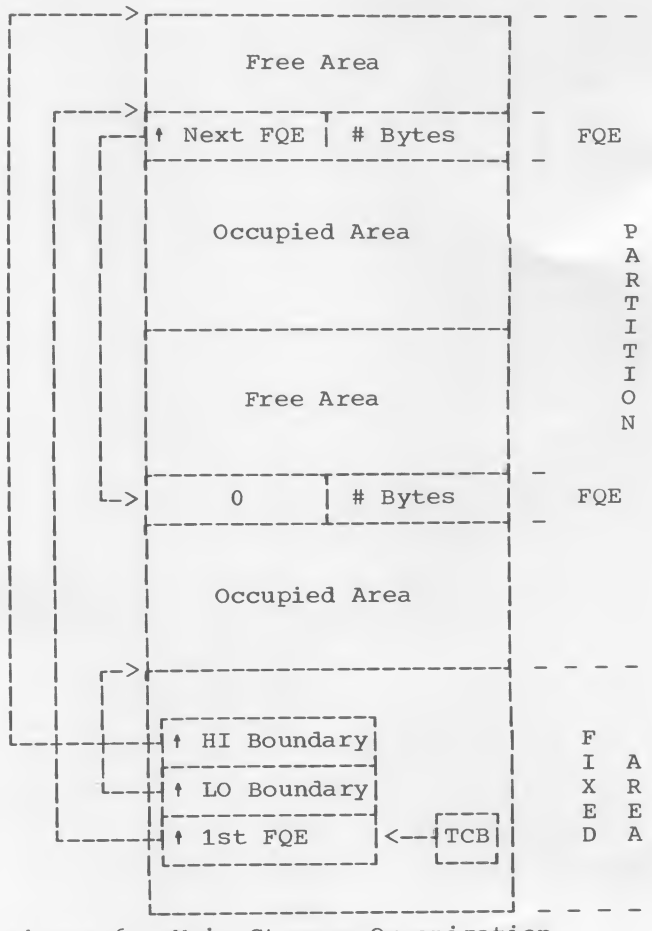


Figure 6. Main Storage Organization

More specifically, when a processing program executes a GETMAIN macro-instruction with a numbered subpool request ranging from 0 through 127, storage is allocated in the upper end of the partition. A request with a subpool number from 128 through 255 is invalid for processing programs, and causes task termination. When a privileged routine executes GETMAIN with a subpool number ranging from 0 through 127, storage is allocated in the lower end of the partition; subpool numbers 128 through 255 cause storage to be allocated in the upper end of the partition. However, by convention, subpool numbers 129 through 248 are not used.

Areas not in use at a given time are referred to as free areas and are represented in a free area queue by a series of free area queue elements (FQEs). Each free area begins and ends on a double word boundary; requests for main storage space are always rounded up to multiples of eight bytes.

Each free area queue element is eight bytes long. The first four bytes contain the address of the next lower free area if there is one, or zero if there is no lower free area. The second four bytes contain the length of the free area. The free area queue element is always in the lowest eight bytes of each free area.

The first element in the free area queue is pointed to by the first word of a three word block in the nucleus. This block, called the boundary box, is initialized by the nucleus initialization program and is pointed to by the TCBMSS field in the TCB. The boundary box contains the address of the beginning of the partition in its second word, and the address-plus-one of the end of the partition in its third word.

GETMAIN

When a GETMAIN is executed, the free area queue is searched for space as large or larger than that required. If found, the space is allocated, and the amount used is subtracted from the *free area* from which it was removed. If space is not found and the request was conditional, GETMAIN ends by branching to type 1 exit. If the area is not found and the request was unconditional, GETMAIN branches to ABTERM to schedule the termination of the task.

In systems with the optional inactive program list, GETMAIN frees all routines on the inactive program list pointed to by the TCB if adequate space is not found by searching the free area queue. GETMAIN returns the space in which the freed routines resided to the free area queue and searches again. GETMAIN always frees the inactive program list whenever a system routine requests space in the lower end of the partition.

FREEMAIN

When a FREEMAIN is executed, the area to be freed is checked for any overlap with existing free areas. If overlap exists, an error has occurred and FREEMAIN branches to ABTERM for the scheduling of an abnormal termination of the task. Otherwise, FREEMAIN combines the area to be freed with any adjacent free area, by updating that area's FQE. If there are no adjacent free areas, FREEMAIN creates an FQE for the newly freed area and queues the FQE on the free area queue. On completion, FREEMAIN branches to type 1 exit.

CHAPTER 4: CONTENTS SUPERVISION SERVICE ROUTINES

Contents supervision service routines record the identity, main storage location, size, properties and users of routines which, with the data they operate on, make up tasks. Completed routines are not immediately destroyed but may remain in storage until the space is required. Contents supervision service routines maintain three lists (see the discussion of request block queueing in the introduction to this manual) of routines in each partition:

- Active request block queue -- a list of active routines given control by type II, III, or IV linkage, excluding type 1 SVCs.
- Inactive program list (optional) -- a list of inactive routines originally brought into storage by LINK, XCTL or ATTACH, but which are no longer in use.
- Loaded program list -- a list of frequently-used routines brought into storage by a LOAD.

Each routine in these lists is represented by an RB that immediately precedes the routine in main storage. Exceptions to this are: the SIRB, which is permanently in the nucleus; SVRBs, which are always in the upper end of main storage, away from their associated routines; and "minors," which are RBs placed on the loaded program list by the optional IDENTIFY macro-instruction and which represent routines embedded in the processing program.

Contents supervision maintains the three lists by chaining together the RBs for the routines. Each list is pointed to by the TCB.

HOW CONTENTS SUPERVISION IS ORGANIZED

Contents supervision is made up of the following service routines: LINK, LOAD, XCTL, IDENTIFY (optional), DELETE, SYNCH, and a common subroutine called FINCH.

The LINK service routine passes control from the routine that issued the LINK macro-instruction to another routine so that the issuer regains control when the second routine completes.

The LOAD service routine brings a routine specified in the parameters of a LOAD macro-instruction into main storage and inserts its RB on the loaded program list with a use count of one. If the routine is already on the list, the service routine merely adds one to the use count, which thus reflects the number of times a LOAD has been issued for this routine minus the number of times a DELETE has been issued for it.

The XCTL service routine passes control from the routine issuing the XCTL macro-instruction to a requested routine. When the requested routine completes, control is not returned to the issuer, which has been removed from the active RB queue, but to the routine which preceded the issuer of the XCTL. The issuer of the XCTL is removed from main storage if it has not been LOADed

The IDENTIFY service routine causes a routine named by the issuer of the IDENTIFY macro-instruction to have a minor RB created for it, and causes this RB to be chained on the loaded program list. The RB which is the result of the IDENTIFY is on the LOAD list only for control purposes. The RBs of these identified routines are removed from the loaded program list and the RB space is released whenever these routines are inactive and the routine containing them is placed on the inactive program list or is deleted.

The DELETE service routine decrements the use count in the RB of a LOADed routine named by the issuer of a DELETE macro-instruction. When the use count becomes zero, DELETE removes the RB from the loaded program list and frees the storage space occupied by the routine. (Note: In systems which include the IDENTIFY macro-instruction, any minors associated with the named routine are also removed by DELETE.)

The SYNCH service routine creates, initializes, and queues program request blocks. System routines or processing programs use this routine to create PRBs for segments of code which they designate by placing an entry point address in register 15 and executing an SVC SYNCH instruction. After the PRB is queued on the active request block queue, SYNCH returns by executing an SVC EXIT instruction.

The FINCH service routine interfaces with the data management BLDL routine, and with program fetch which is described in the next chapter of this manual, to retrieve routines from auxiliary storage. Routines may be retrieved when a LINK,

LOAD, XCTL, or ATTACH macro-instruction is issued, or when a non-resident SVC routine or non-resident input/output supervisor error routine is requested. After the routines are loaded into main storage, FINCH records information concerning their attributes and main storage locations into the appropriate contents supervision lists.

CONTENTS SUPERVISION CONTROL FLOW

As shown in Chart 04, the flow of contents supervision is essentially the flow of the individual service routines, which receive control from interruption supervision and pass control to their particular exit routine on completion. FINCH is an exception in that it receives control from LINK, LOAD, and XCTL, as well as from a number of other system routines including ATTACH and the SVC FLIH, and returns to whatever routine requested its services.

LINK

The LINK service routine is entered by the SVC SLIH in response to a LINK macro-instruction.

LINK searches the loaded program list for the RB of the requested routine and if it is found and is inactive, prepares the RB for dispatching. If the routine is not found or if it is active, LINK checks the first RB on the inactive program list. If this RB represents the requested routine, and is reschedulable, LINK prepares the RB for dispatching.

When these two steps fail, LINK clears the inactive program list and frees the storage occupied by the represented routines, and enters FINCH. FINCH constructs an RB for the requested routine and places both the RB and its routine in the lower end of the partition.

On return from FINCH, LINK prepares the RB for dispatching by:

- Initializing LINK's SVRB so that register loading causes the requested routine to execute EXIT when it issues the *RETURN* macro-instruction.
- Flagging the requested routine's RB to indicate that it is active.
- Placing the requested routine's RB on the active RB queue between the RB for LINK and the RB for the issuer of the request, to ensure that the requested routine is entered when LINK issues EXIT.
- Issuing the SVC EXIT instruction.

LOAD

The LOAD service routine is entered by the SVC SLIH when a LOAD macro-instruction is issued. LOAD searches the loaded program list for the RB of the requested routine, and if it finds it, increments the use count and passes the requested routine's entry point to the issuer in register 0. LOAD branches to the terminal portion of LINK that issues the SVC EXIT instruction.

If the requested routine is not found on the loaded program list, LOAD branches to FINCH to load the routine into storage. On return from FINCH, LOAD initializes the requested routine's RB and places it on the loaded program list, sets the RB's use count to one and branches to LINK to issue the SVC EXIT instruction.

XCTL

The XCTL service routine is entered by the SVC SLIH when an XCTL macro-instruction is issued.

If XCTL was issued by an SVC routine operating in the transient area, the XCTL service routine branches to FINCH to locate the routine on the SVC library and bring it into the transient area. XCTL branches to that part of LINK that completes the initialization of the RB and executes an SVC EXIT instruction.

If XCTL was not issued by a transient routine, the XCTL routine dequeues the issuer's RB and its minors from the active RB queue. The RB for the routine which issued XCTL is placed on the inactive program list unless it was LOADed. If the requested routine is on the loaded program list and inactive, XCTL branches to LINK to:

- Set the active bit in the RB for the requested routine.
- Queue the RB on the active RB queue.
- Issue an SVC EXIT instruction.

If the RB of the requested routine was not found inactive on the loaded program list, XCTL frees storage of the routines

represented on the inactive program list and branches to FINCH to bring in the routine. On return from FINCH, XCTL initializes *the routine* in the same manner as if its RB had been found inactive on the loaded program list.

IDENTIFY

The IDENTIFY service routine is entered by the SVC SLIH in response to the issuance of an IDENTIFY macro-instruction which is an option in the fixed-task environment.

IDENTIFY builds and initializes a minor request block to describe a routine specified in the parameters of the IDENTIFY macro-instruction, and chains this minor to the loaded program list and to the RB of the routine which contains the identified routine. IDENTIFY returns to the issuer by issuing an SVC EXIT instruction.

DELETE

The DELETE service routine is entered by the SVC FLIH when a DELETE macro-instruction is issued. The DELETE routine decrements the use count in the RB of the routine specified in the parameters of the DELETE macro-instruction. If the use count reaches zero, DELETE dequeues the routine from the loaded program list and issues a FREEMAIN macro-instruction to release the storage occupied by the specified routine and its RB. On return from FREEMAIN, *DELETE* repeats the deleting process for any minors belonging to the specified routine. DELETE returns by branching to the type 1 SVC exit.

SYNCH

The SYNCH service routine is entered by the SVC SLIH when a SYNCH macro-instruction is executed. SYNCH uses GETMAIN to obtain 32 bytes of main storage from the lower end of the partition for the creation of a PRB. The PSW in the PRB is initialized by SYNCH to address the location specified in register 15 by the issuer of the macro-instruction. SYNCH sets the PSW completely enabled in problem program mode, with the protection key recorded in the task control block. After the PRB is created and initialized, SYNCH queues it on the active request block queue below the SVRB for SYNCH, and returns by issuing an SVC EXIT instruction.

COMMON SUBROUTINE (FINCH)

The FINCH service routine is entered by a branch from seven other system routines and it returns to them by a branch. The seven service routines which branch to FINCH are

- ATTACH
- LINK
- LOAD
- XCTL
- SVC SLIH
- EXIT EFFECTOR
- EXIT

FINCH uses the data management BLDL routine to locate a named routine on an external storage device. Using the information provided by BLDL, FINCH initializes the program fetch parameters and uses the program fetch routine to bring the specified routine into main storage. FINCH allows for necessary RBs when issuing GETMAIN, and initializes them with the RB type and the size of the storage space they and their routines occupy.

Program fetch, a part of the resident nucleus, places into main storage load modules obtained from the system library or any other library organized as a partitioned data set. Program fetch is reenterable; that is, it can be used concurrently by more than one task. The module name of program fetch is IEWFTMIN.

Program fetch has two entry points. Contents supervision passes control to program fetch by branching to IEWMSEPT, overlay supervision passes control to program fetch by branching to IEWFBOSV.

A load module is placed into main storage using block loading, which places an entire load module into a contiguous main storage area. IEWFTMIN operates only in the block loading mode.

HOW PROGRAM FETCH IS ORGANIZED

Program fetch is organized to perform the following specific functions:

- Initialization. Performs initialization procedures to prepare for the loading of a module.
- Loading. Reads text records and RLD records of a load module into main storage.
- Relocation. Adjusts values of address constants to reflect the relocation of a module that has been loaded into main storage.
- Termination. Performs termination procedures after a module has been loaded into main storage.

PROGRAM FETCH CONTROL FLOW

Program fetch receives control from contents supervision when either a LINK, ATTACH, LOAD, or XCTL macro-instruction is issued and a usable copy of the module specified is not in main storage. When contents supervision requests a block module, program fetch loads the entire module. A load module with the scatter attribute is block loaded. When an overlay module is requested, only the root segment is loaded.

Program fetch receives control from overlay supervision when a segment of an overlay program specifies another segment that is not in main storage either by a branch or by issuing a SEGWT or CALL macro-instruction. After receiving control from overlay supervision, program fetch loads the requested segment.

The initialization procedures shown in Chart 05 are performed each time program fetch begins execution. Control then passes to the loading routine, which reads in the module. Relocatable address constants embedded in text records are adjusted by the relocation routine. Control passes between the loading routine and the relocation routine until the entire segment or module is loaded and relocated. Termination procedures are then performed and control is returned to the caller.

INITIALIZATION

Contents supervision supplies program fetch with the following parameters (see program listing for contents of general registers and fetch parameter list):

- Main storage address of applicable partitioned organization directory record.
- Main storage address of an opened data control block (DCB) to be used while loading the module.
- Main storage address of the work area to be used (see Figure 7).
- Main storage address of area into which NOTE list is to be read for overlay programs (see Figure 8).
- Main storage address at which to begin loading the module.
- Return address in general register 14.

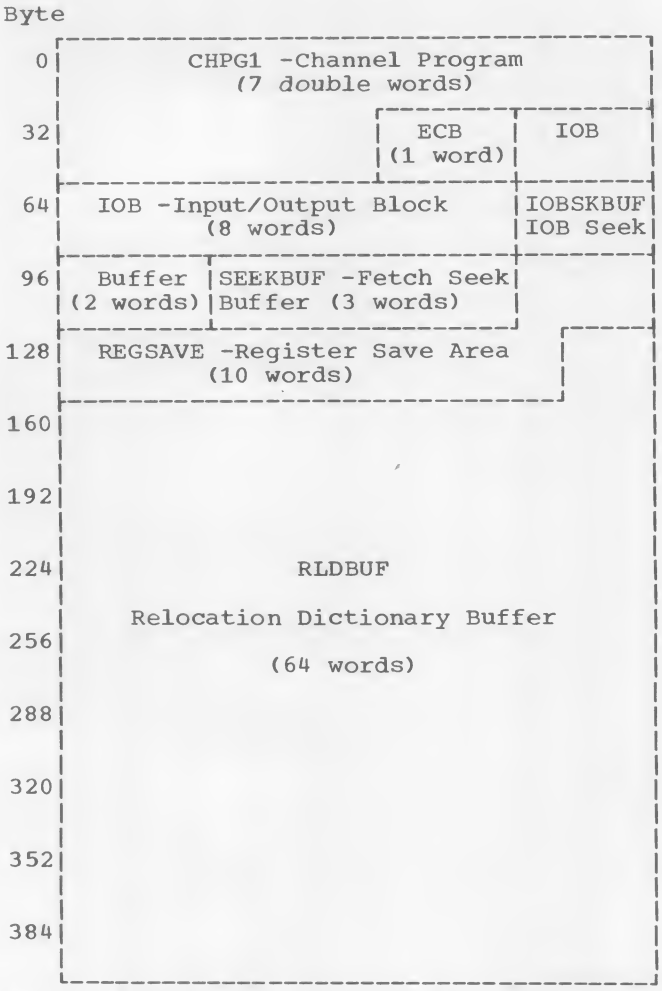
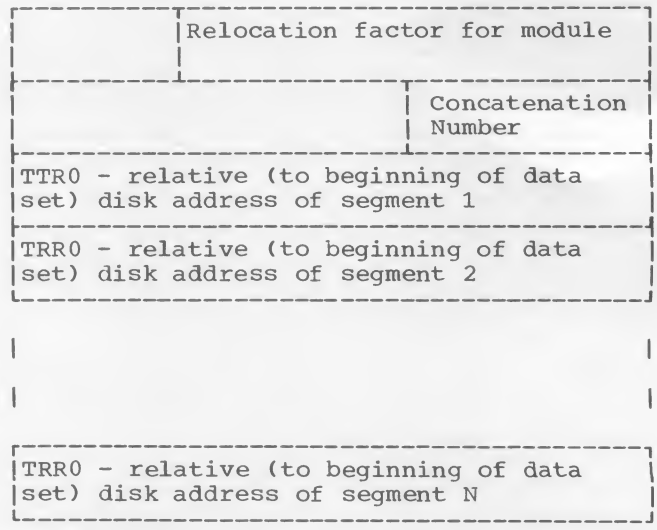


Figure 7. Program Fetch Work Area

- Main storage address of a work area for use by program fetch.
- Segment number of the requested segment multiplied by 4.
- Return address in general register 14.



Concatenation Number - This a value specifying this data set's sequential position within a group of concatenated data sets.

Figure 8. Note List (in Main Storage)

Overlay supervision supplies program fetch with the following parameters:

- Main storage address of the data control block (DCB) previously used to read in the root segment.
- Main storage address of the note list (loaded before the root segment).

After receiving control, program fetch uses the parameters supplied to build an input/output block (IOB), an event control block (ECB), and a channel program (CCW list) in the specified work area. The channel program is used to read in the program, and if necessary, the note list containing the relative disk addresses of the overlay module's segments. Figure 9 shows the relationship of the blocks and tables used by program fetch to load block and overlay modules.

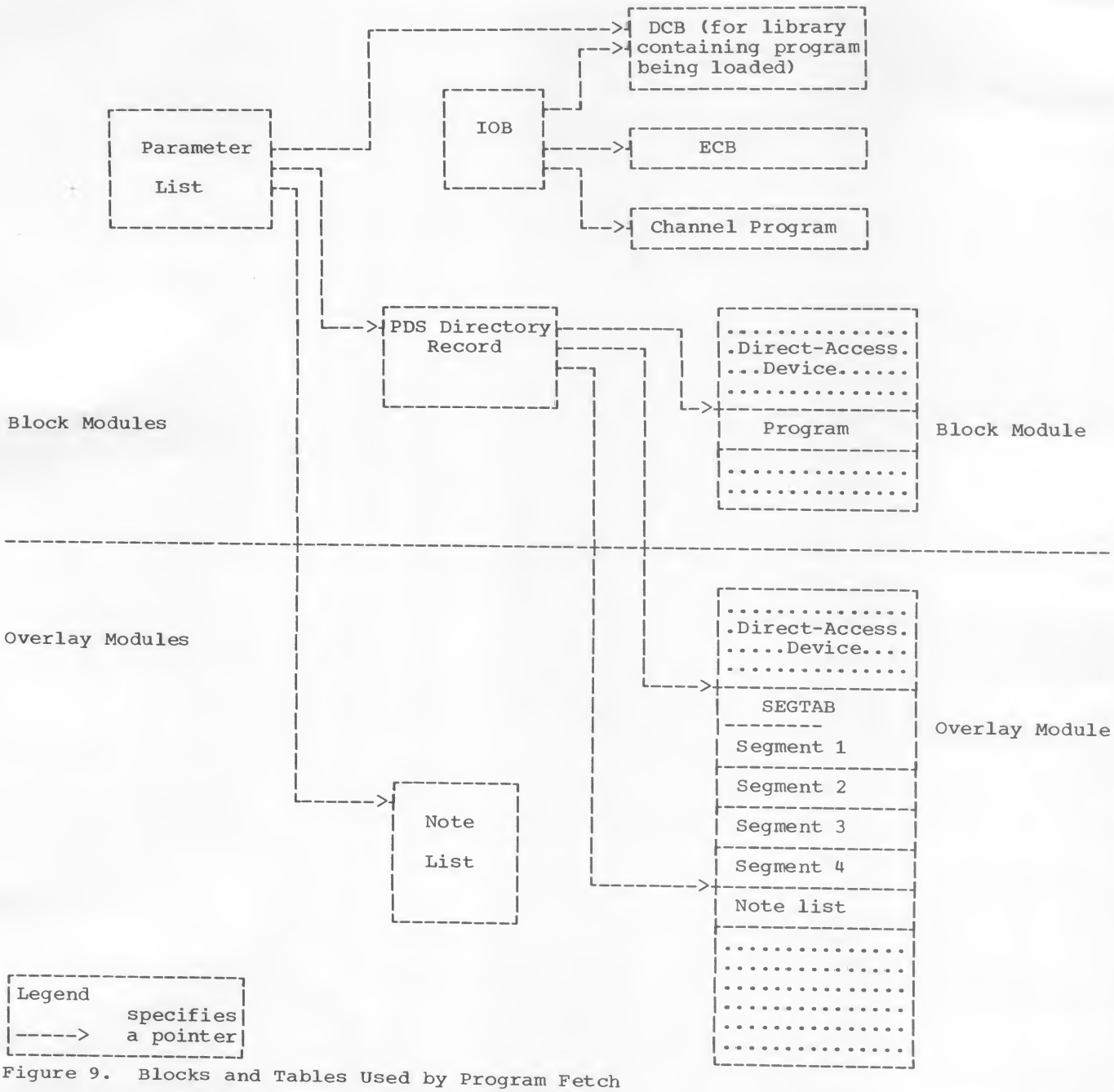


Figure 9. Blocks and Tables Used by Program Fetch

LOADING

A load module (Figure 10) consists of control records, text records, RLD records, and composite control RLD records. These records are of variable length. Their formats are shown in Appendix D.

After control is received from contents supervision, program fetch obtains the length and the relative disk address of a module's first text record from the partitioned organization directory record (see Appendix D). Subsequent text records are read using the length given in the control record preceding each text record. One or more records containing RLD information will follow a text record that has embedded relocatable address constants. Program fetch uses the RLD information to find and adjust the values of the address constants.

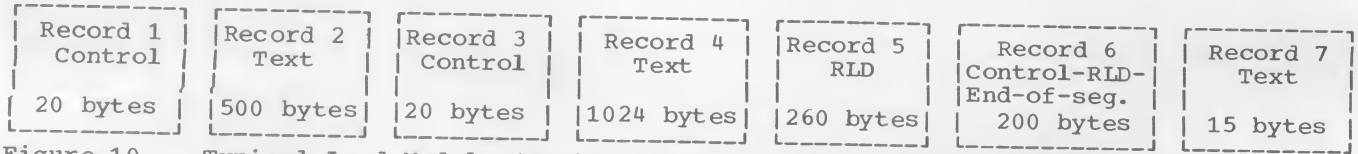


Figure 10. Typical Load Module (Logical Format on Direct-Access Device)

When loading a block or overlay module, program fetch alters the mode of its channel program according to the type and sequence of records contained in the module (see Figure 11). The normal sequence of records in a module is: control information - text record - control information - text record. Two records are read at a time as long as the normal sequence -- a text record followed by control information -- is encountered. When the second of the two records read in the normal mode does not contain control information, program fetch alters the mode of the channel program so that a subsequent EXCP macro-instruction causes a single record to be read. Each record read singly is checked for control information. If present, program fetch restores its channel program to the normal mode. Text records are read into their assigned main storage location; RLD records are read into the RLD buffer.

As program fetch loads a module, it reads the count record preceding each data record into the fetch seek buffer. The channel program's search command specifies the last count record read. This is the count record that precedes the last data record that was read. When the count record specified by the search command is found, a subsequent read count, key and data command will result in skipping the data record that followed the count record and will begin reading at the next count record, as shown in Figure 12.

Program fetch causes a single record to be read by turning off the command chaining bit in the first read CCW of the channel

Condition	Number of Records Read With Each EXCP Issued	Source (if any) of Record Length and Relative Disk Address (TTR), if not reading sequentially
Normal first EXCP for all modules including root segment of overlay modules	2	Partitioned Organization Directory Record
Normal Mode	2	Control record provides record length of following text record
First EXCP for a segment	1	NOTE list provides relative disk address (TTR)
EXCP for a NOTE list	1	Partitioned Organization Directory Record
EXCP to read a control and/or RLD record that previously caused an incorrect length input/output error	1	None
Previous record was RLD only (did not contain control information)	1	None
EXCP for a module that consists of one text record and no RLD records	1	Partitioned Organization Directory Record
Last record of the module is a text record	1	Control record provides record length of following text record.

Figure 11. Conditions Affecting Channel Program Mode

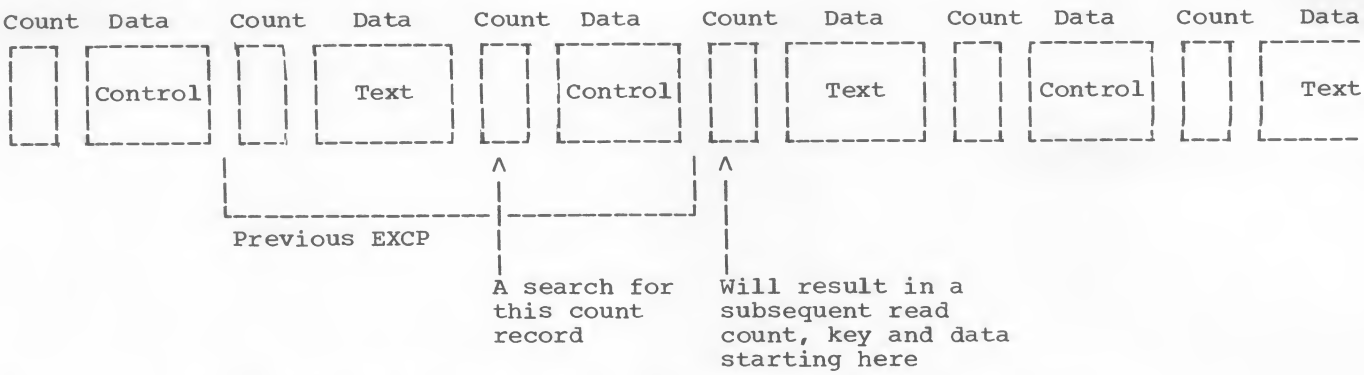


Figure 12. Typical Load Module (Physical Format on Direct-Access Device)

program when either of the following conditions occur:

- The last text record of a module is to be read (indicated by the setting of the end-of-segment bit in the preceding control record).
- A module to be loaded consists of a single text record without any RLD information following it (indicated by the module's attributes in the PDS directory).

Overlay Modules

When an overlay module is loaded, its NOTE list is first read into main storage. The root segment is then read into main storage using normal block loading procedures.

While an overlay program is being executed, the NOTE list which contains the main storage address of the SEGTAB and the relative disk addresses of the module's segments, remains in main storage.

After the root segment has been loaded the SEGTAB is initialized. Program fetch inserts, into SEGTAB, the main storage address of data control block (DCB) and the NOTE list, and if required, sets the TESTRAN indicator.

To read in a segment other than the root segment, program fetch uses a relative disk address found in the NOTE list to read the first control record of the segment. The information in the control record is used to begin reading in the segment in the normal mode.

End-of-Extent Appendage

A load module may reside in one or more extents on a direct-access device. The boundaries of these extents are specified in the data extent block (DEB) for the library containing the module being loaded. When an EXCP macro-instruction is issued that results in crossing one of the extent boundaries within which a portion of the module being loaded resides, the input/output supervisor passes control to program fetch's end-of-extent appendage. The appendage acquires the beginning extent boundary for the next portion of the load module from the DEB, places it into the unit control block (UCB), and returns control to the input/output supervisor.

Input/Output Errors

All input/output errors are handled by the I/O supervisor, except incorrect length errors occurring while reading control and/or RLD records.

Normally, an incorrect length indication is expected when reading control and/or RLD records, since they are variable length and their specific length is not known in advance. After reading such a record with a maximum possible count (256 bytes), program fetch examines the content of the record to check that what was read was of correct length. If this check fails, program fetch makes one more attempt to read the record, this time with the exact expected count. If the attempt to reread fails, control is given to the caller and an error code is passed.

RELOCATION (ADJUSTING ADDRESS CONSTANTS)

Program fetch adjusts address constants by adding (or subtracting) a relocation factor to (or from) the address constant's value that is embedded in the load module.

When a module is block loaded, its relocation factor is the difference between its linkage editor assigned address, which is always zero, and the first byte of main storage into which the module is to be loaded. For example, assume a module is to be loaded into main storage beginning at address 4000. If the RLD flag bit is positive a relocation factor of +4000 is added to the relocatable address constant. If, however, the RLD flag bit is negative, the relocation factor is subtracted from the address constant (see Appendix D for RLD entry format). The linkage editor assigned address of every relocatable address constant is given by the relocation dictionary (RLD).

Address constants in the root segment of an overlay module are adjusted in the same manner as those in a block module. The root segment's relocation is used to adjust the address constants of all segments of the module since an overlay module is essentially block loaded. The relocation factor is stored in the NOTE list by program fetch and is available throughout the execution of the overlay module.

TERMINATION

When a block module or the root segment of an overlay module has been loaded, program fetch computes the relocated entry point of the module and places it in the fetch (parameter) list. When a root segment of an overlay module is loaded, program fetch also inserts the main storage address of the data control block (DCB) and the NOTE list into the segment table (SEGTAB).

To specify a successful or unsuccessful loading, program fetch passes the appropriate termination code to its caller. Control is then returned to the caller via a branch to the address in the link/return register.

The overlay supervision service routines control the loading of overlay program segments and assist the flow of control between the segments of an overlay program. While performing these functions, these routines place data into and use data from the segment table (SEGTAB) and the entry tables (ENTABs).

Because the segment and entry tables are part of each overlay program, the overlay supervisor is reenterable and its services can be used concurrently by many overlay programs.

During execution, an overlay program issues requests for segments. The requests can be explicit via a SEGLD or SEGWT macro-instruction or implicit via a branch that is routed through an ENTAB. In either case, the overlay supervisor receives control from the SVC handler and checks the SEGTAB to determine whether the requested segment is in main storage. If not, the overlay supervisor requests program fetch to load the segment. When this segment is part of an overlay program that is being tested, the overlay supervisor also passes control to the TESTRAN interpreter.

Program fetch and the TESTRAN interpreter each return control to the overlay supervisor after their functions have been performed.

SEGLD is not supported in this configuration; a SEGLD request is treated as a NOP instruction.

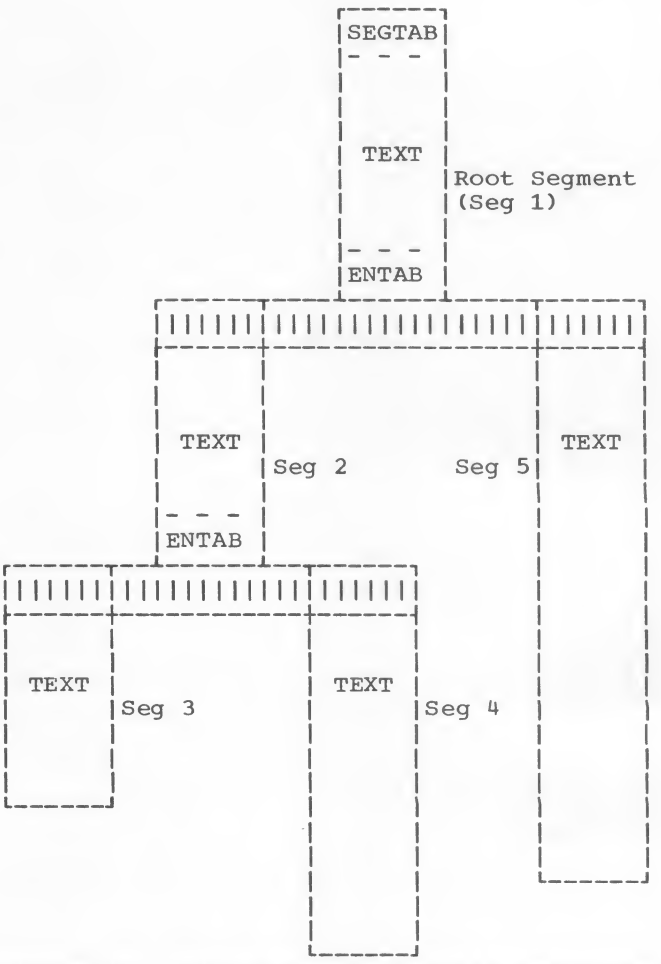


Figure 13. Single-Region Overlay Structure

TABLES USED BY OVERLAY SUPERVISION

The segment table (SEGTAB) and the entry tables (ENTABs) that contain the data used by the overlay supervisor are created by the linkage editor from information in the relocation dictionary (RLD) and the user's control statements.

Figure 13 shows the SEGTAB and ENTABs in a typical single region overlay structure; the ENTAB and SEGTAB formats are given in Appendix E.

USE OF SEGMENT TABLE

The segment table (SEGTAB) contains data that describes the structure and status of an overlay module, and is a directory for the segments of that module. It contains both fixed and variable information. The fixed information includes:

- TEST indicator. This indicator is set by program fetch if the partitioned organization directory record indicates that the program is being tested under TESTRAN.

- Last segment number of each region. This value defines the segment that ends a region and is used to determine the region that contains a particular segment.

- Previous segment number of each segment in the module. The overlay supervisor uses this field to determine the additional segments that must be loaded with the requested segment. (These additional segments are those in the path of the requested segment.)

The variable information includes:

- Pointers. These pointers are addresses of the NOTE list and DCB.

- Highest number segment of each region in main storage. This value is initialized to 1 for the first region by the linkage editor.

- Status indicator for each segment. The overlay supervisor sets a status indicator for each segment to indicate either that the segment is not in main storage, that the segment is being loaded into main storage, or that the segment is present in main storage.

For more information about the SEGTAB, see Appendix E.

USE OF ENTRY TABLES

The entry tables (ENTABs) assist in passing control between the overlay supervisor and an overlay program. They handle downward branches in an overlay program, that is, the branches to segments lower in the path.

When the overlay program executes an upward branch, the overlay supervisor is not entered, and the ENTABs and SEGTAB are not used. An upward branch is direct because segments in the path are always in main storage (Figure 14).

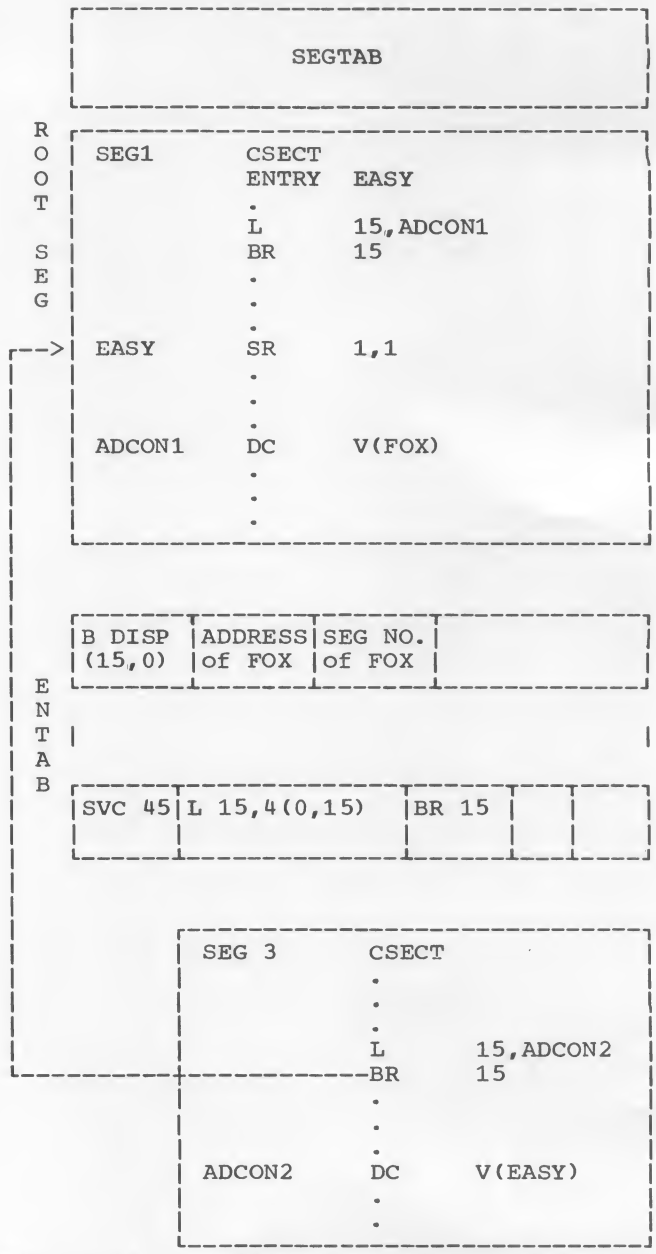


Figure 14. Overlay Program Upward Branch

Branching to a Segment Not in Main Storage

When an overlay program branches to a segment not in main storage, control is passed to the applicable ENTAB (step A of Figure 15). The branch instruction in the ENTAB passes control to an SVC instruction contained in the first field of the last ENTAB entry (step B). The SVC instruction

causes an SVC interruption, and passes control to the SVC handler and then to the overlay supervisor (step C). The overlay supervisor uses a pointer in general register 15 to obtain the information required to:

- Determine the number of the requested segment from the ENTAB.
- Determine the status of the requested segment from the SEGTAB.
- Pass control to the requested segment at the entry point specified by the address of the entry point field in the ENTAB.

After the segment is loaded, control is returned to the second field of the last ENTAB entry, the instruction following the SVC (step D). When the load and branch instructions have been executed, control is passed to the correct entry point.

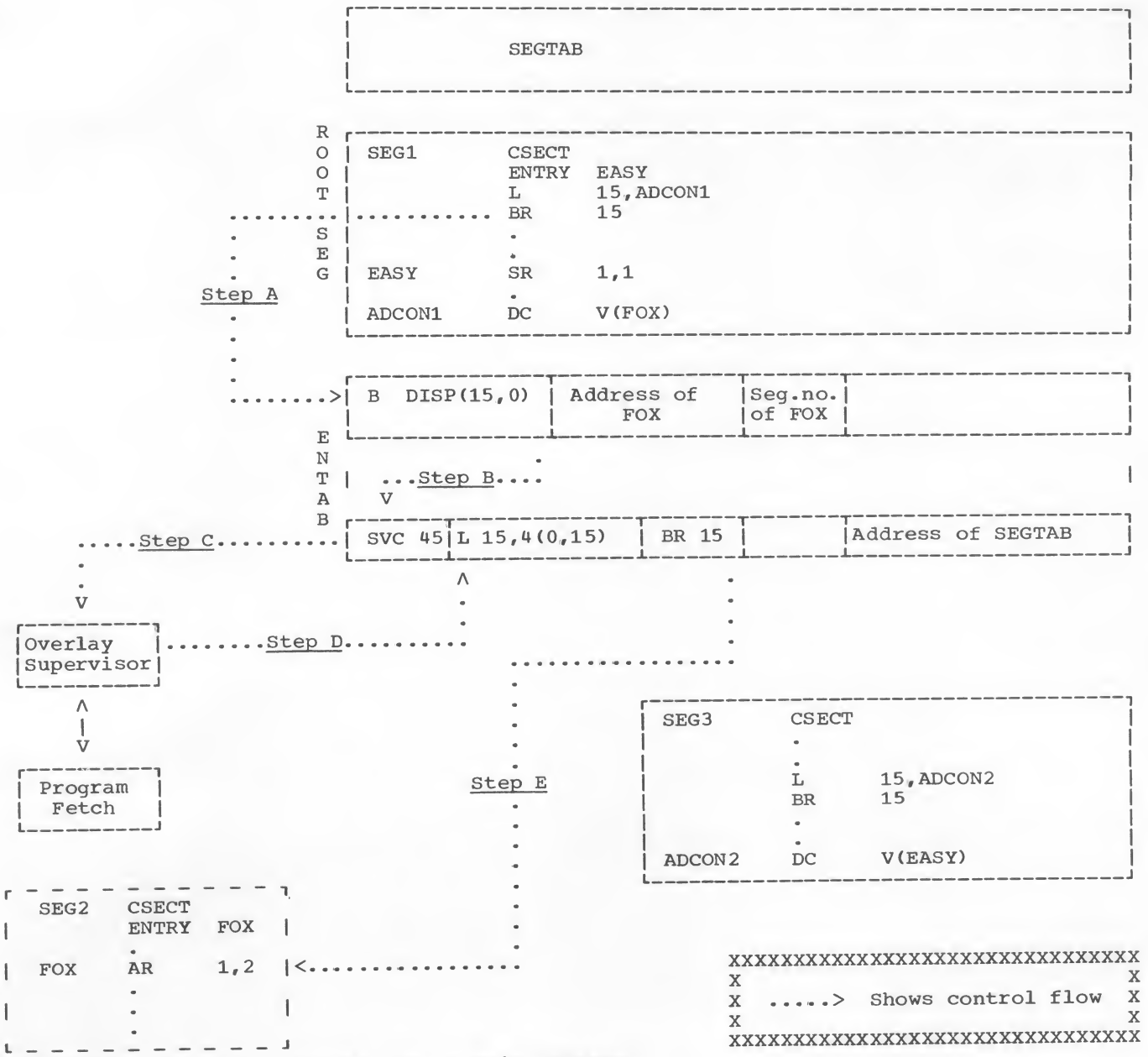


Figure 15. Branch to Segment Not in Main Storage

Branching to a Segment in Main Storage

When a segment is loaded into main storage, because of an implicit call (a branch through an ENTAB), the displacement (DISP) field in the ENTAB entry through which the branch was routed is increased by 2 (Figure 16). When the overlay program executes another branch to this ENTAB entry, the SVC instruction is bypassed, and control is given to the second field of the last ENTAB entry. Execution of the instruction in this field causes general register 15 to be loaded with the main storage address assigned to the indicated symbol. A branch to that location is then executed.

A caller is an ENTAB entry that assisted in routing a branch from a segment to an entry point in a segment lower in the path. ENTAB entries that have been modified to bypass the SVC instruction are chained together in a caller chain (Figure 17). These entries are chained only if the called and calling segments are located in the same region. Chaining is accomplished by placing a pointer to (address of) the modified ENTAB entry into the caller field of the SEGTAB when the segment is brought into main storage. If this segment is requested again, the contents of the SEGTAB caller field (a pointer to a previous caller) is placed into the previous caller field of the referred to ENTAB entry, and a pointer to this ENTAB entry is placed in the caller field of the SEGTAB. In this way, a chain is created that begins at the SEGTAB entry and points to all the ENTAB entries (in the same region) that were modified (+2) to bypass the SVC 45 instruction. When the segment is to be overlaid, the caller chain is used to reset all of the modified ENTAB entries in the chain.

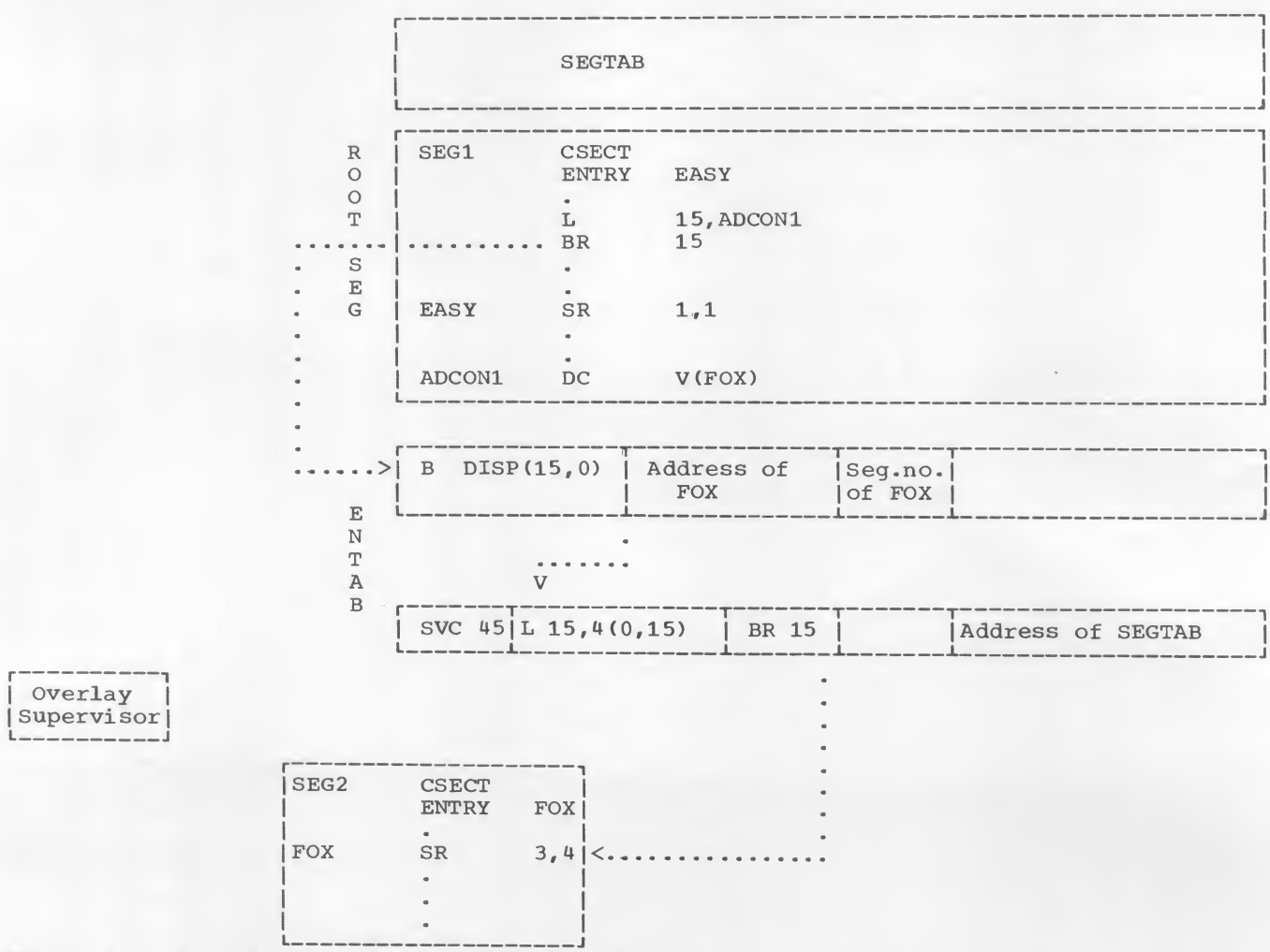


Figure 16. Branch to Segment in Main Storage

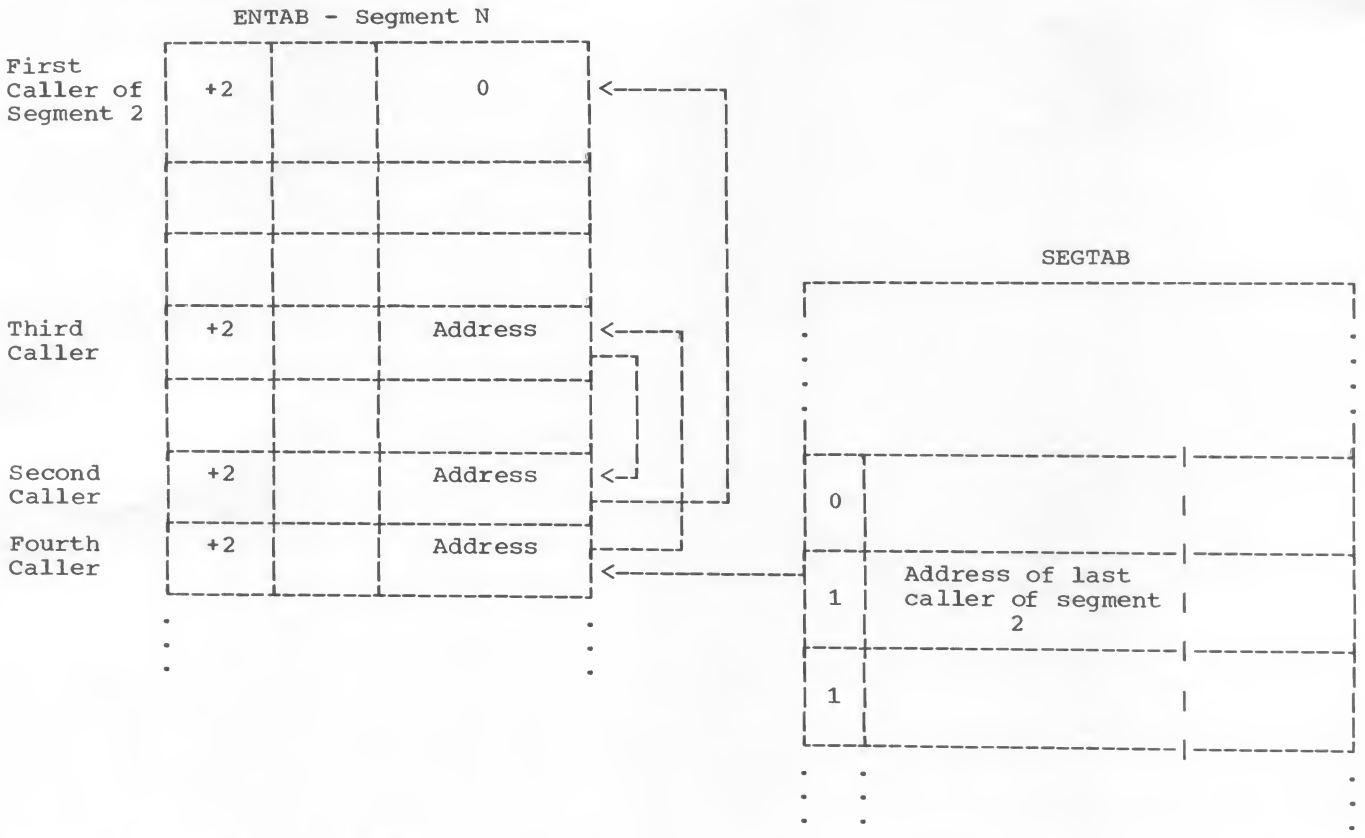


Figure 17. Chaining of ENTAB Entries Used to Branch to a Segment

HOW OVERLAY SUPERVISION IS ORGANIZED

Overlay supervision is composed of a resident module called overlay supervisor 1 and either of two non-resident modules selected at SYSGEN time called overlay supervisor 2.

The module name of overlay supervisor 1 is IEWSVOVR; the module name of overlay supervisor 2 is IEWSYOVR for the basic synchronous module or IEWSXOVR for the basic synchronous module with optional SEGWT checking. To pass control to either version of overlay supervisor 2, overlay supervisor 1 issues a LINK macro-instruction that specifies IEWSZOVR, which is the member name of the selected module in the LINKLIB.

OVERLAY SUPERVISION CONTROL FLOW

The resident module has two entry points: IGC037 and IGC045. The SVC handler passes control to IGC037 as a result of an SVC 37 instruction (SEGWT macro-instruction), or to IGC045 as a result of an SVC 45 instruction (an intersegment branch that is routed through an ENTAB). An SVC 37 instruction with zero in general register 0 specifies a SEGLD macro-instruction, whereas a one in general register 0 specifies a SEGWT macro-instruction. (SEGLD is treated as a NOP in a single-task environment.) Chart 06 shows overlay supervisor control flow.

Overlay supervisor 1 is permanently resident in the nucleus of the operating system. It performs the first portion of initialization and then links to overlay supervisor 2. When control is returned to overlay supervisor 1, it performs the remaining termination procedures and issues an SVC EXIT instruction.

When a requested program is an overlay program, contents supervision issues a LOAD macro-instruction to bring overlay supervisor 2 into main storage. Overlay supervisor 2 remains in main storage for the duration of the task that required it. When given control by overlay supervisor 1, overlay supervisor 2 performs the remaining initialization procedures, loads the requested segments, updates the segment

table (SEGTAB) and entry tables (ENTABs), performs some termination procedures, and then returns control to overlay supervisor 1.

INITIALIZATION

During linkage editor processing, if the address constants of a segment are resolved to an ENTAB, the number of the segment is placed in the high-order byte of the address constants. The V-type address constants that are not resolved to an ENTAB contain a zero in their high-order bytes. The address constants can be the result of an expansion of a SEGLD, SEGWT, or CALL macro-instruction, or the result of the user creating an address constant for use with a branch instruction. If a SEGLD or SEGWT request is received and the high-order byte of the V-type address constant is zero, the request is treated as a NOP.

The overlay supervisor obtains the segment number of the requested segment from the "to segment number" field in the ENTAB. The overlay supervisor obtains the address of the SEGTAB from the last entry in the ENTAB, and checks the SEGTAB to determine the segment's status and relationship to the overlay structure.

The basic synchronous module with optional checking (IEWSXOVR) detects overlay requests that would cause the requesting segment to be overlaid. This module checks only those requests that result from the execution of a SEGWT macro-instruction.

UPDATING OF TABLES

Before segments are loaded, the overlay supervisor updates the SEGTAB and ENTABs of the overlay program to reflect the changes to be made in the overlay structure present in main storage. For each segment that is logically overlaid, a status indicator is reset in the SEGTAB. The SEGTAB is scanned to find the caller chains (Figure 15), which are used to reset the ENTAB entries to their original state (the state before the segment containing the corresponding entry point was loaded into main storage). The ENTAB entries are reset by subtracting +2 from the displacement field of the branch. When the SEGTAB and ENTAB entries of the last segment have been updated, the segments are loaded.

SEGMENT LOADING

During segment loading, the overlay supervisor scans the SEGTAB to determine which segments are needed and directs program fetch to load the requested segment and all segments in its path that are not in main storage.

TERMINATION

The overlay supervisor checks the TEST indicator in the SEGTAB to determine if the overlay program is "under test". If under test, a LINK macro-instruction is issued specifying the TESTRAN interpreter. After TESTRAN interpreter execution, control is returned to overlay supervisor.

If the overlay supervisor was entered via an SVC 45 instruction (through an ENTAB), and the ENTAB through which the request was routed is in the root segment or is in the same region as the requested segment, the caller chain is updated (Figure 15) and the address field of the branch is altered in the calling ENTAB. If the requesting and requested segment are not in the same region, the caller chain and the branch instruction in the ENTAB are not altered. Subsequent branches to an altered ENTAB entry are routed directly to the segment.

Control is returned to overlay supervisor 1.

The time supervision service routines are an optional feature of the fixed-task supervisor for installations that have selected the hardware timer as a part of their Computing System/360. Time supervision processes requests for the date and time of day, and requests for setting a time interval interruption, for checking if that interval has elapsed, and for cancelling that interval. Additional functions include maintaining a queue of pending requests and maintaining the relationship between the actual time of day and the hardware.

HOW TIME SUPERVISION IS ORGANIZED

Time supervision is made up of the following service routines: timer second level interruption handler (SLIH), STIMER, TIME, and TTIMER.

The timer SLIH monitors all types of interval expirations, including those of the control program, and maintains the queue of time interval requests.

The STIMER service routine sets an interval into a software interval timer, specifies when that interval timer is to be decremented and what action is to be taken when an interruption signals completion of the interval. It does these things in response to an STIMER macro-instruction.

The TIME service routine places the time of day in register 0 and the current date in register 1, when requested through a TIME macro-instruction. The time returned is the time of day based on a 24-hour clock that is set with local time by the operator through the SET command.

The TTIMER service routine tests the interval timer in response to a TTIMER macro-instruction, and places in register 0 the time remaining in the TASK or REAL interval previously set by an STIMER macro-instruction. The TTIMER service routine can also cancel previously specified intervals.

THE TIMING ALGORITHM

Within the timer SLIH is a 4-byte field called the 6-hour pseudo clock (SHPC). By manipulating the values contained in the SHPC and the hardware timer, time supervision maintains real time while timing a prespecified interval.

For example, assume that the 6-hour time of day (TOD), defined as equal to the contents of the SHPC minus the contents of the hardware timer, is zero hours. A request is received for a one hour interval. This is accomplished by placing one hour in the SHPC and in the timer.

```
SHPC - timer = 6-hour TOD
1 hour - 1 hour = 0 hour
```

After an hour, the contents of the timer have automatically decremented to zero and an interruption occurs.

```
SHPC - timer = 6-hour TOD
1 hour - 0 hour = 1 hour
```

If a 2-hour interval is requested, two hours is added to the timer and two hours is placed in the SHPC.

```
SHPC - timer = 6-hour TOD
(1 hour + 2 hours) - 2 hours = 1 hour
```

Two hours later, when the interruption occurs, the correct 6-hour TOD of three hours is indicated by the SHPC.

To correlate the internal, software pseudo clock time with real time, two other pseudo clocks are maintained by time supervision. One is a 24-hour pseudo clock called the T4PC. The other is a local time pseudo clock called the LTPC.

Each time the SHPC reaches six hours the SHPC is reset to zero and six hours is added to T4PC. The T4PC is reset to zero each time 24 hours pass. The T4PC is initially set to zero at initial program load. The contents of the T4PC plus the 6-hour TOD is defined as the T4PC TOD.

The contents of the LTPC initially is equal to the time keyed in at the console by the operator through the SET command. The local time of day which is returned, when requested, is computed by adding the contents of the LTPC to the T4PC TOD.

The three basic time relationships of the timing algorithm are:

- The 6-hour TOD is equal to the contents of the 6-hour pseudo clock minus the contents of the hardware timer.

- The 24-hour TOD is equal to the contents of the 24-hour pseudo clock plus the 6-hour TOD.
- The local TOD is equal to the contents of the local time pseudo clock plus the 24-hour TOD.

Time supervision maintains a queue (Figure 18) of timer queue element (Figure 19) representing interval requests. The timer queue is a two-way chain ordered so that the request for the next interruption is at the top of the queue, while the request for the last interruption is at the bottom of the queue. To ensure that the timer queue element is inserted at the right place in the queue when a new request is received, the interval requested is translated into a value that is relative to the software clocks. This is done by adding the value of the interval requested to the 6-hour TOD. This new value is placed in the TQVAL field of the timer queue element and is used by the queueing subroutine of the timer SLIH to position the element on the queue.

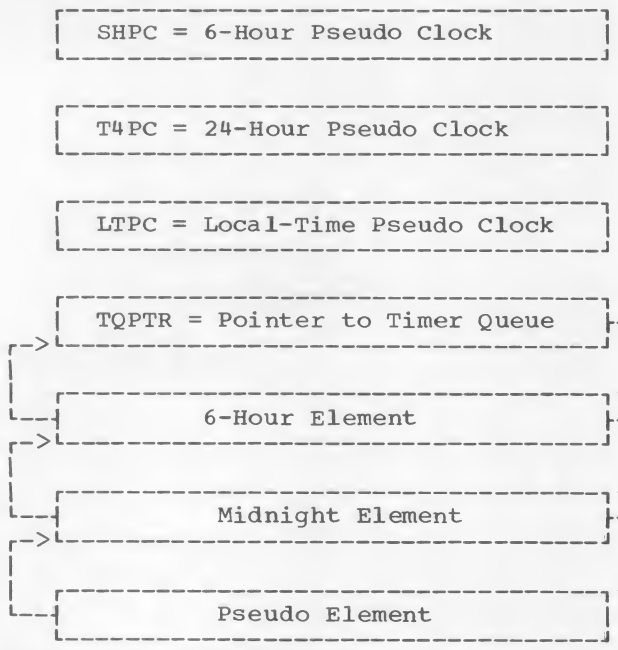


Figure 18. Timer Queue

When the element reaches the top of the queue, the interval placed in the timer is calculated by subtracting the value of the contents of the SHPC from the value of the contents of the TQVAL field of the element. The result of this subtraction is added to the timer, while the unsubtracted value of the contents of the TQVAL field of the element is placed in the SHPC.

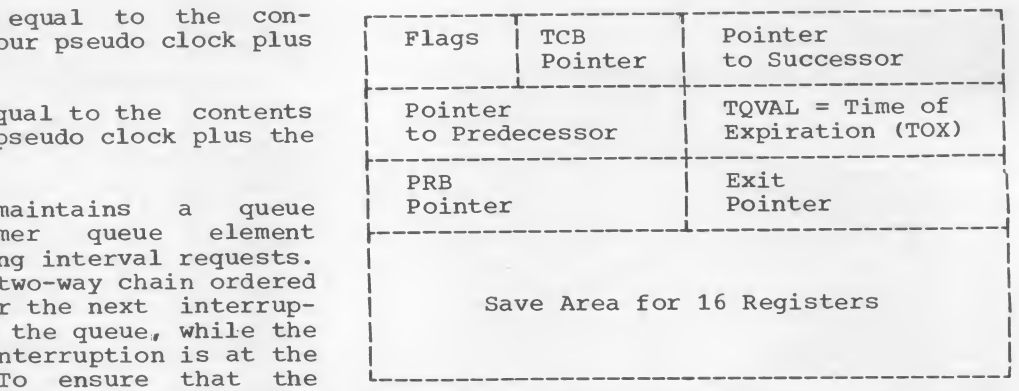


Figure 19. Timer Queue Element (96 Bytes)

At initial program load, two permanent entries are placed on the timer queue representing time supervision interval requests. One is a 6-hour interval request and the other is a request for an interval that is calculated to cause an interruption at midnight, local time. When the midnight interruption occurs, time supervisor increments by one the day-of-the-year count obtained from the operator's SET command. When the six-hour interruption occurs, time supervision updates the T4PC and decrements by six hours the contents of the TQVAL field in each of the elements in the timer queue. In addition, a pseudo element is placed at the end of the queue to mark the queue's terminal point.

TIME SUPERVISION CONTROL FLOW

As shown in Chart 07, the flow of time supervision is generally through two paths. In the first path, control is received from the SVC FLIH by one of the three SVC routines -- STIMER, TIME, and TTIMER. STIMER and TTIMER interface with the timer SLIH's queueing and dequeueing subroutines. TIME and TTIMER return by branching to the type 1 SVC exit, while STIMER executes an SVC EXIT instruction. In the second path, control is received from and returned to the T/E FLIH by the timer SLIH by branching.

STIMER

The STIMER service routine sets up time intervals, represented by timer queue elements, at the completion of which a timer/external interruption will occur. When entered, STIMER initializes the timer queue element's fields. STIMER uses the queueing subroutine of the timer SLIH to

insert the newly created timer queue element into the timer queue. If a WAIT interval is requested, STIMER executes an SVC WAIT instruction.

TIME

The flow through the TIME service routine consists of testing the input parameters of the TIME macro-instruction for the existence of the various options.

The time -- whether formatted in 26-microsecond timer units, ten-millisecond binary units, or packed decimal form -- is always given in terms of local time of day (LTOD). This is calculated according to the formula

LTOD = LTPC + T4PC + SHPC-timer

where LTPC is the contents of the local time of day pseudo clock, T4PC is the contents of the 24-hour pseudo clock, SHPC is the contents of the 6-hour pseudo clock, and timer is the contents of the hardware timer at location 80.

The local time of day is placed in register 0, and the day of the year in register 1.

TTIMER

The TTIMER service routine determines how much time remains in an interval requested by a previous STIMER macro-instruction, and cancels the interval if the CANCEL parameter is present.

When entered, the TTIMER routine determines whether the interval has expired. If it has, no action is taken. If it has not, the time remaining in the tested interval is returned to the user in register 0. TTIMER tests for the cancel option and, if it is present, TTIMER uses the dequeueing subroutine of the timer SLIH to take the timer queue element off the timer queue.

TIMER SLIH

The timer SLIH receives control from the T/E FLIH when a timer interruption occurs. The SLIH identifies the type of interval that has expired and then satisfies the specific requirement.

The SLIH removes the expired timer queue element from the timer queue through one of its two major subroutines (the dequeueing subroutine) resets the hardware timer to time the next interval on the queue, and resets the SHPC. The action taken by the SLIH after an expiration depends on the interval type:

- If it is a WAIT type, the SLIH executes the SVC POST instruction.
- If it is a REAL or TASK type, and an exit address was specified, the exit is scheduled through the Exit Effector routine.
- If it is a 6-hour time supervision type, six hours is subtracted from the TQVAL field of each timer queue element, and the 6-hour interval request is queued again.
- If it is a midnight time supervision type, the day-of-the-year count is incremented by one and the midnight interval request is queued again.

Queueing Subroutine

The queueing subroutine of the timer SLIH is used by the dispatcher, the SLIH, STIMER, and by the SET command handler of job management, to place a timer element on the timer queue. The dispatcher uses the routine when placing a task with a time interval request in control of the CPU.

The queueing subroutine converts the absolute time interval in the element to a relative time based on the 6-hour TOD. If the interval is found to be smaller than the current interval on the queue, the new smaller interval is added to the timer and placed in the SHPC. If the interval is not smaller, the correct insert point on the queue is located for the element, which is queued.

Dequeueing Subroutine

The dequeueing subroutine is used by the dispatcher, STIMER, and TTIMER to remove elements from the timer queue by pointer manipulation. If the element was at the top of the queue, control is passed to the SLIH, which resets the timer and SHPC. Control is passed back to the caller by a branch, at the completion of the dequeueing subroutine, unless a branch was made to the SLIH, which returns control directly to the caller.

Chart 00. Fixed-Task Supervisor Control Flow
(Described in the introduction to this manual)

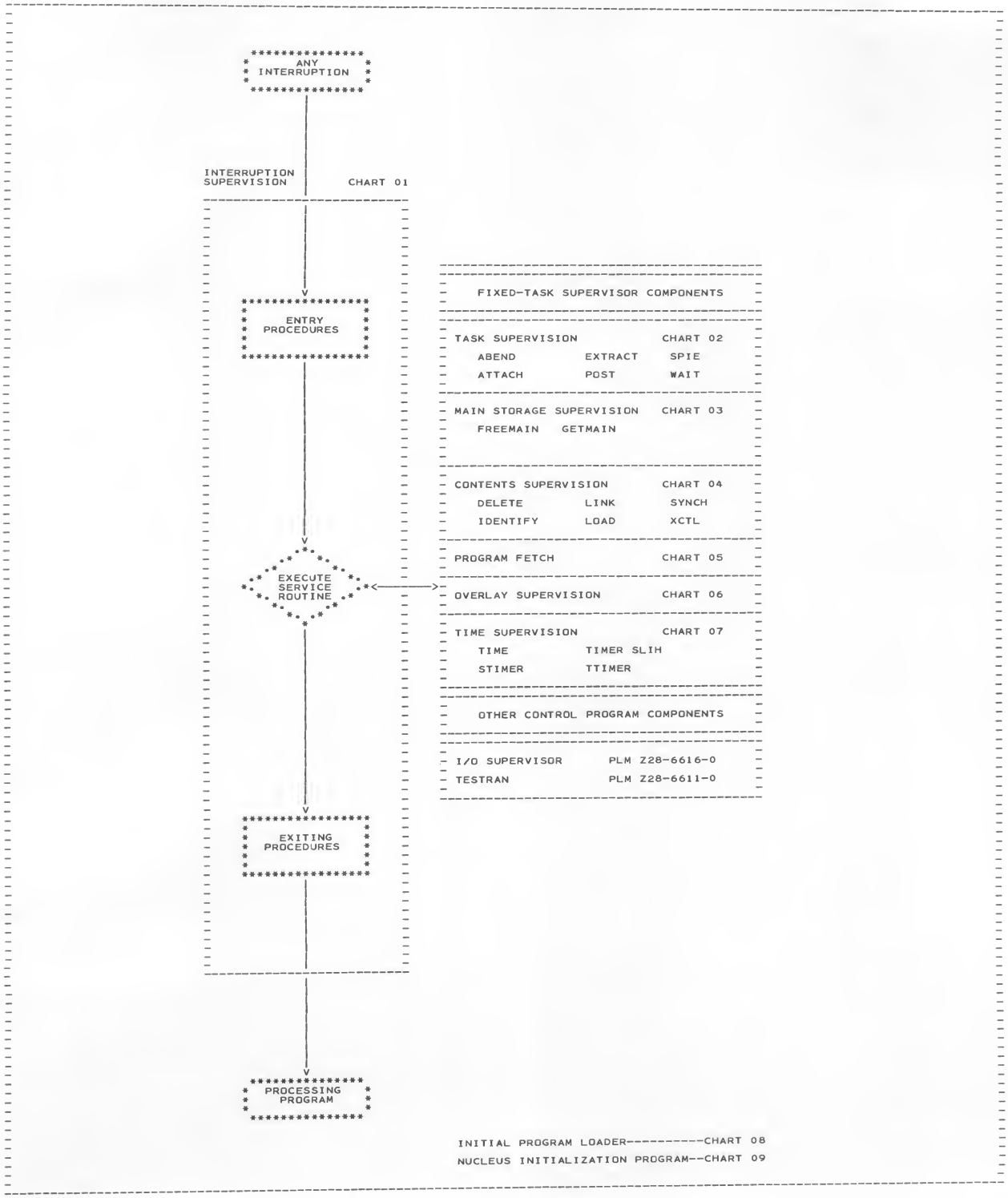


Chart 01. Interruption Supervision Control Flow
(Described in Chapter 1)

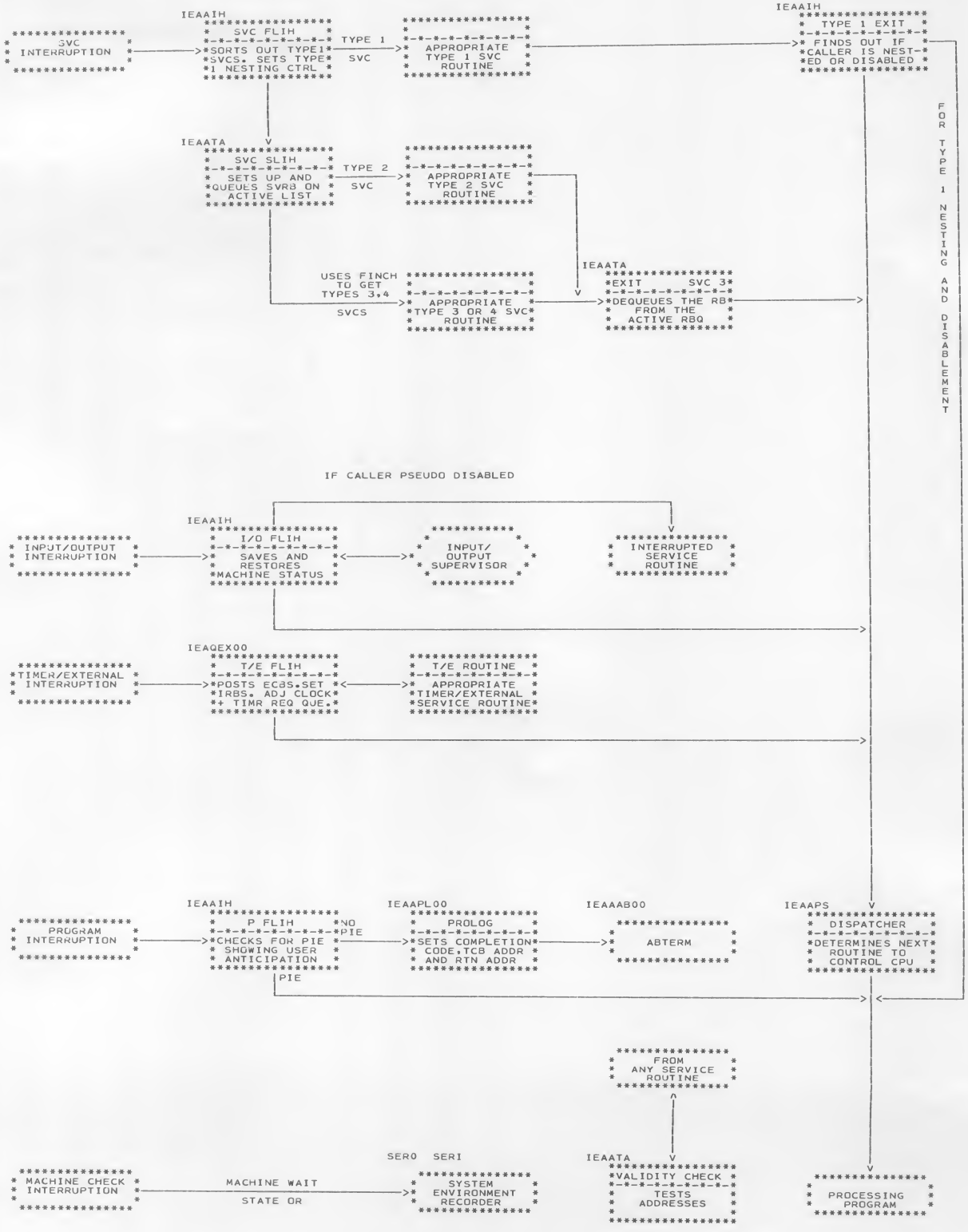
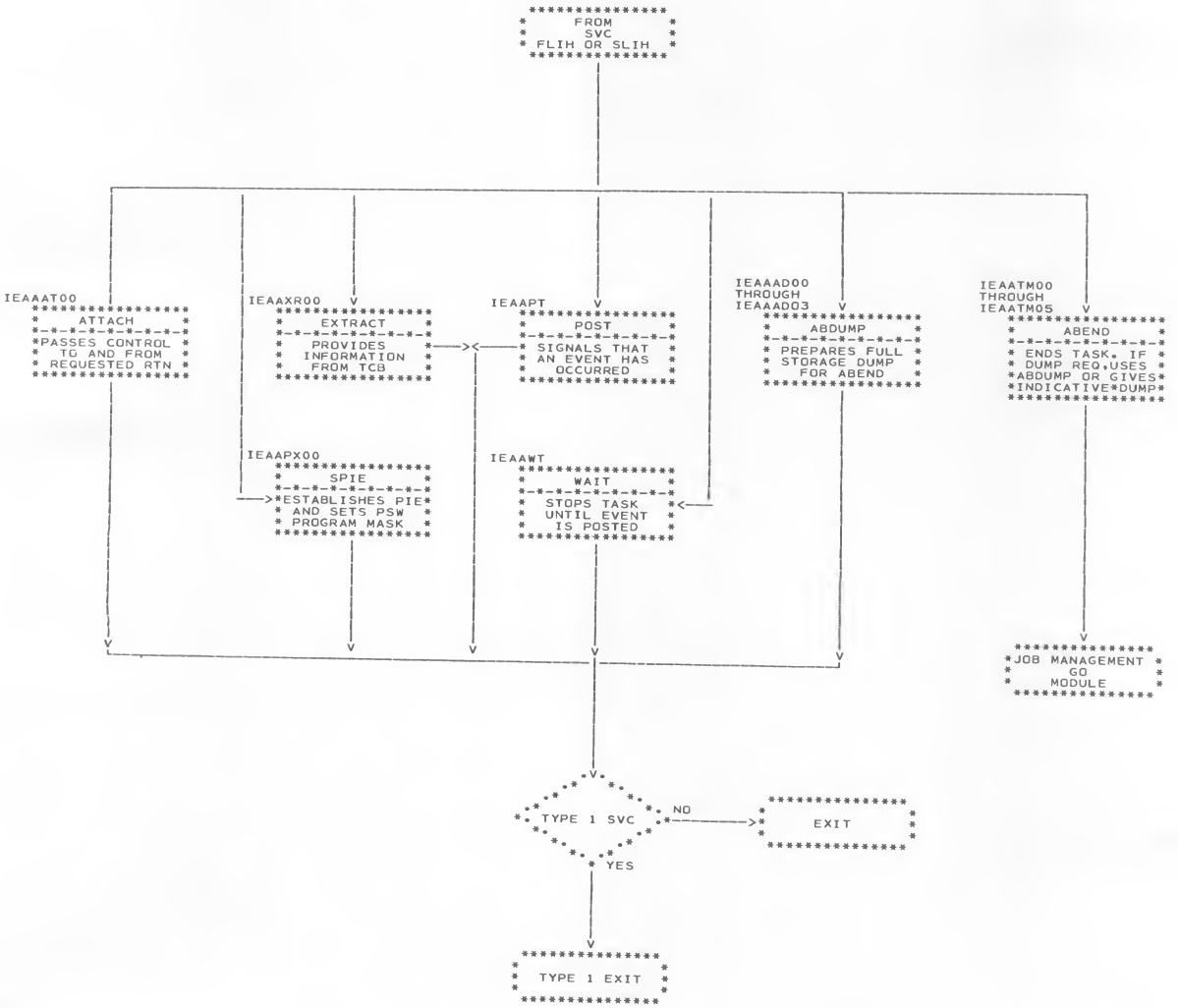


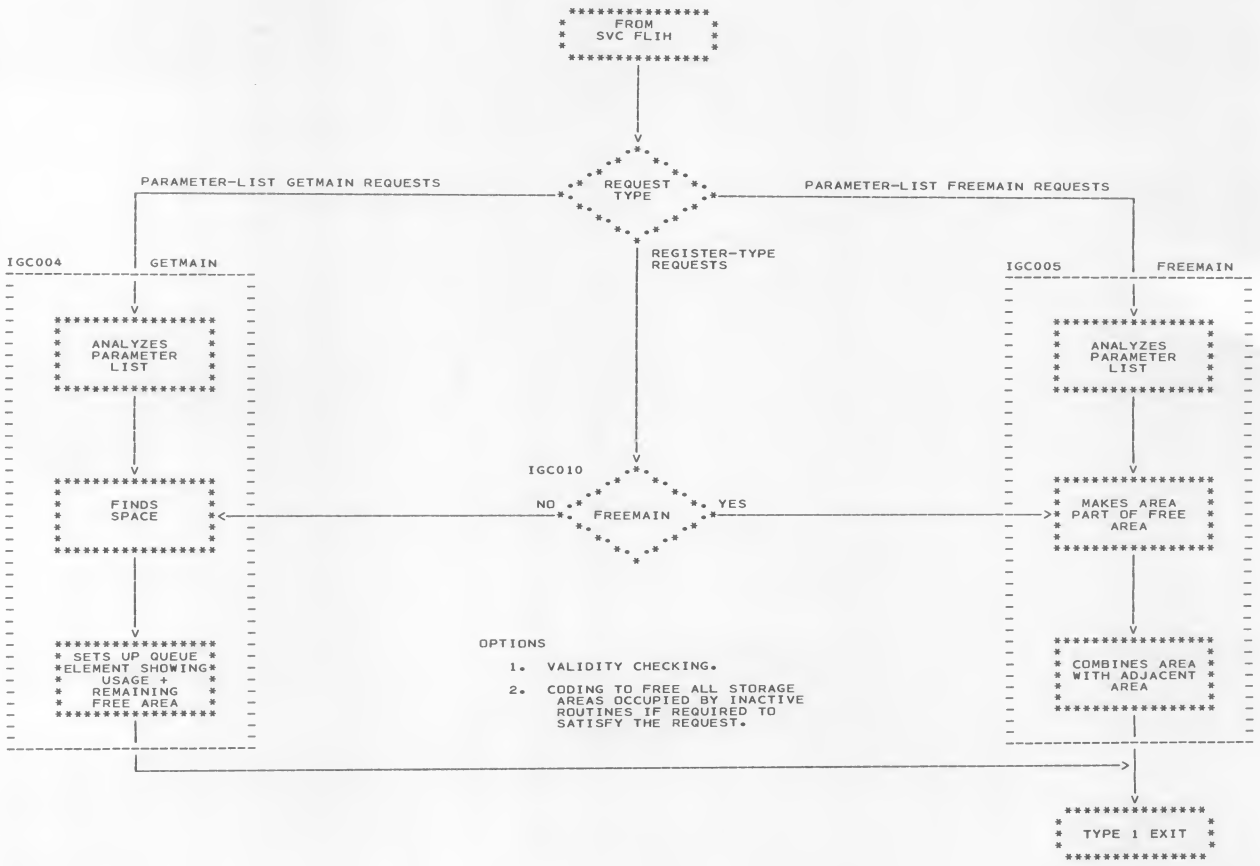
Chart 02. Task Supervision Control Flow
(Described in Chapter 2)



SVC ENTRY AND EXIT PROCEDURES ARE SHOWN ON CHART 01

Chart 03. Main Storage Supervision Control Flow (Described in Chapter 3)

FOR MODULES *IEAAMS00*, IEABMS00, IEACMS00, IEADMS00



SVC ENTRY AND EXIT PROCEDURES ARE SHOWN ON CHART 01

Chart 04. Contents Supervision Control Flow
(Described in Chapter 4)

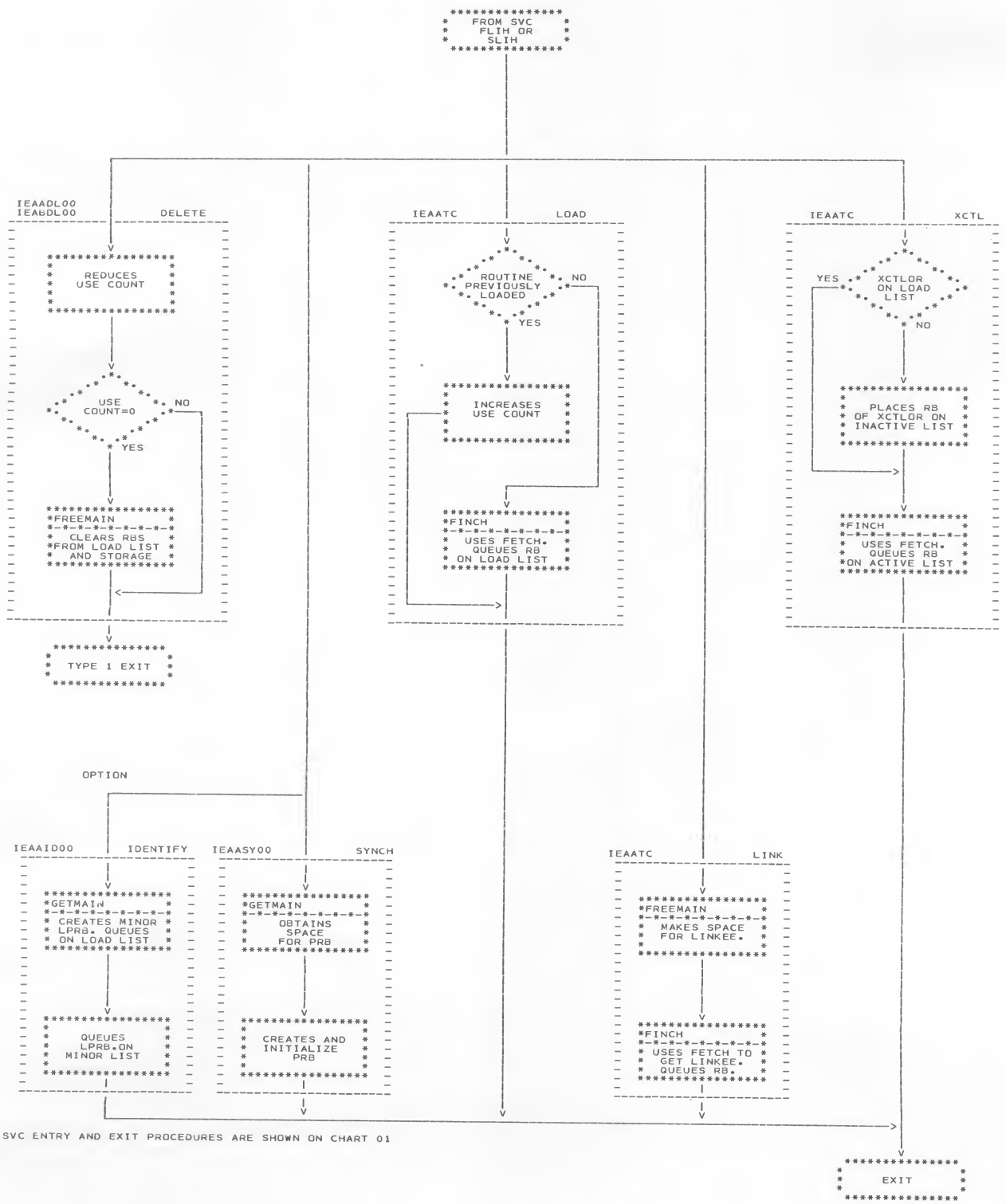


Chart 05. Program Fetch Control Flow
(Described in Chapter 5)

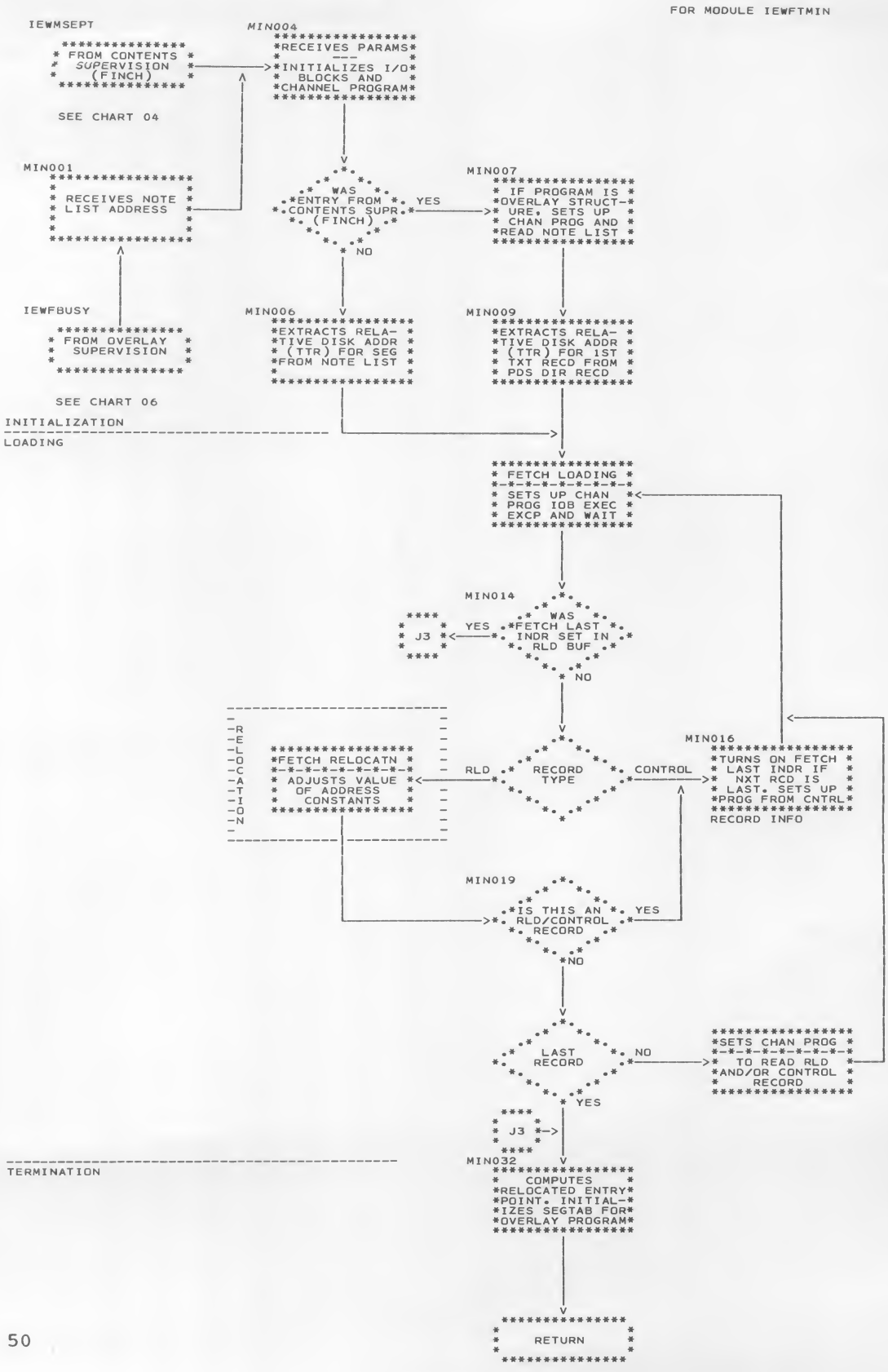


Chart 06. Overlay Supervision Control Flow
(Described in Chapter 6)

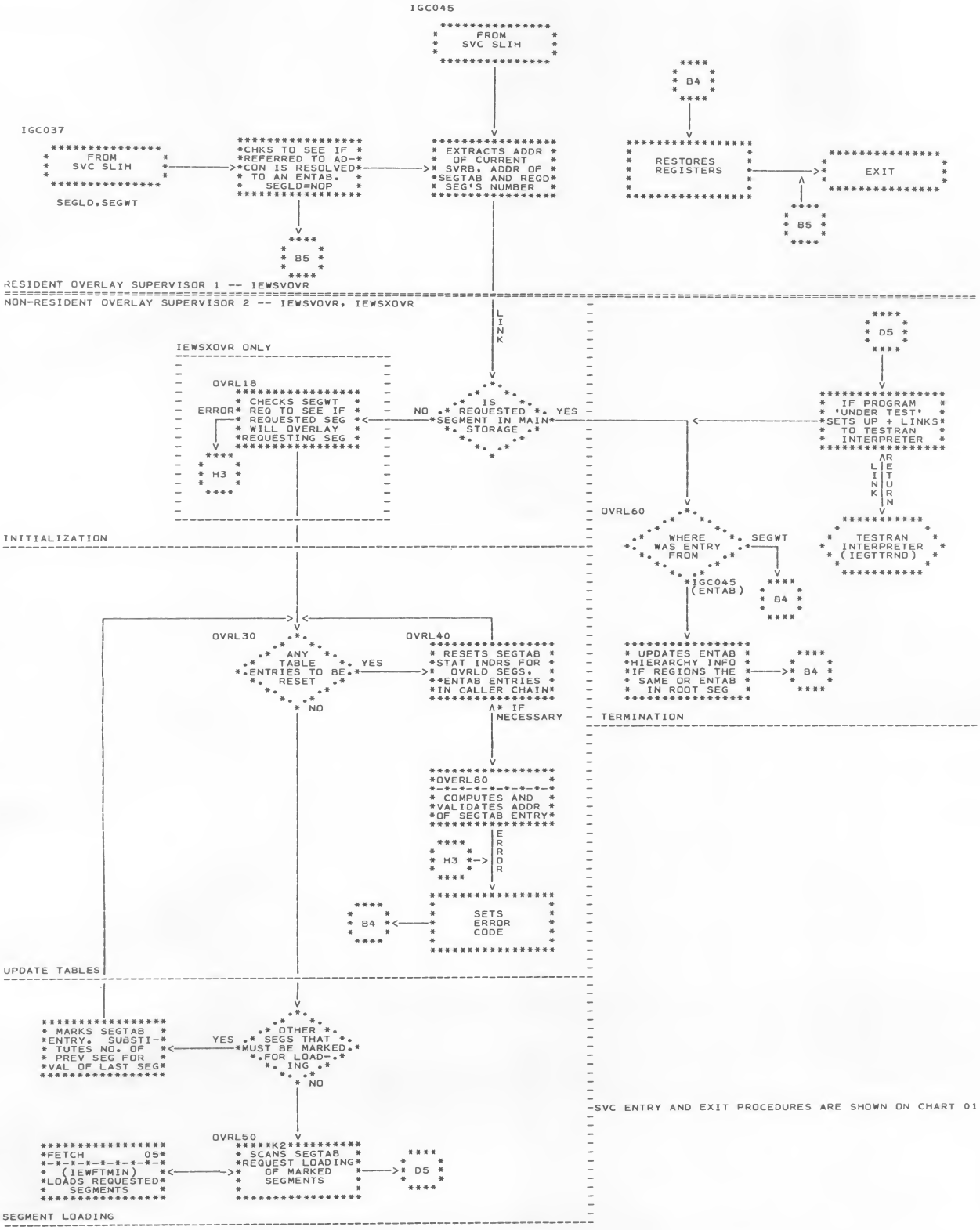
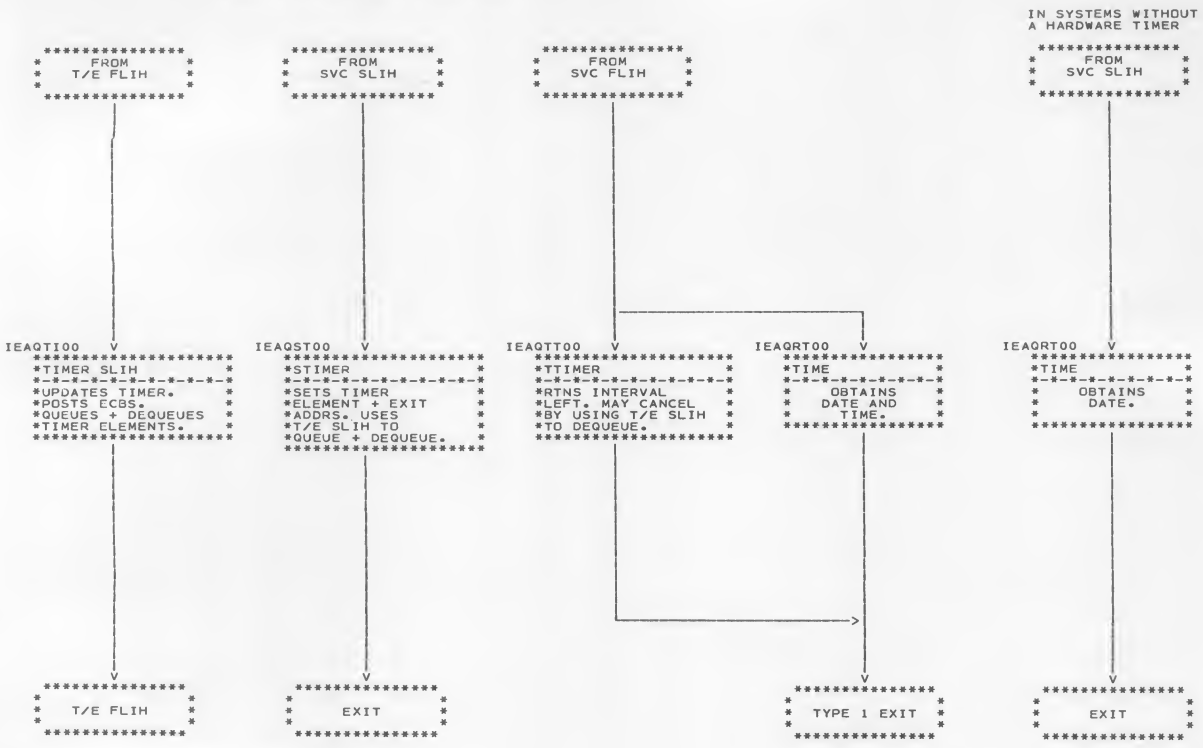


Chart 07. Time Supervision Control Flow
(Described in Chapter 7)



SVC ENTRY AND EXIT PROCEDURES ARE SHOWN ON CHART 01

Chart 08. Initial Program Loader Control Flow
(Described in Appendix A)

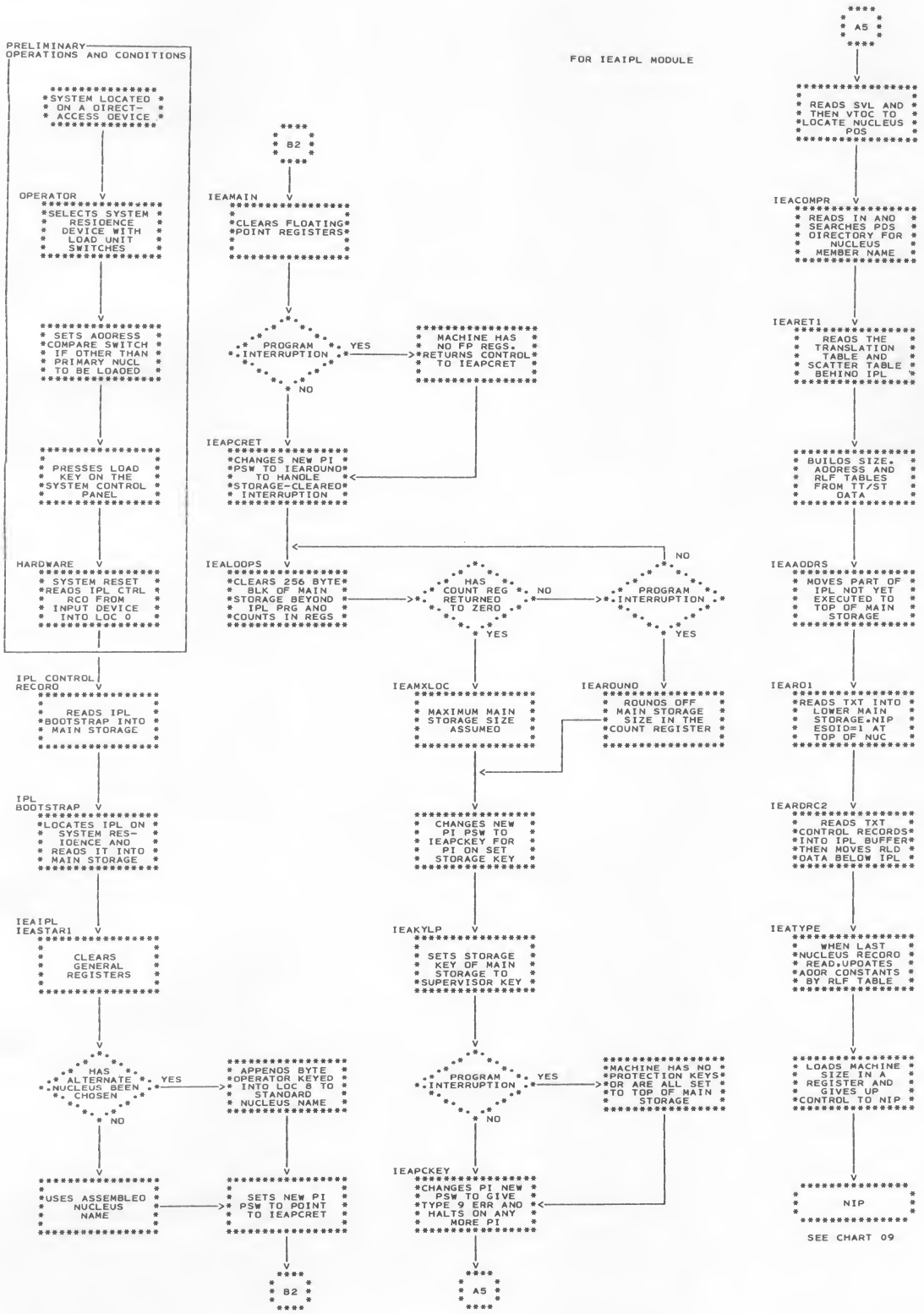
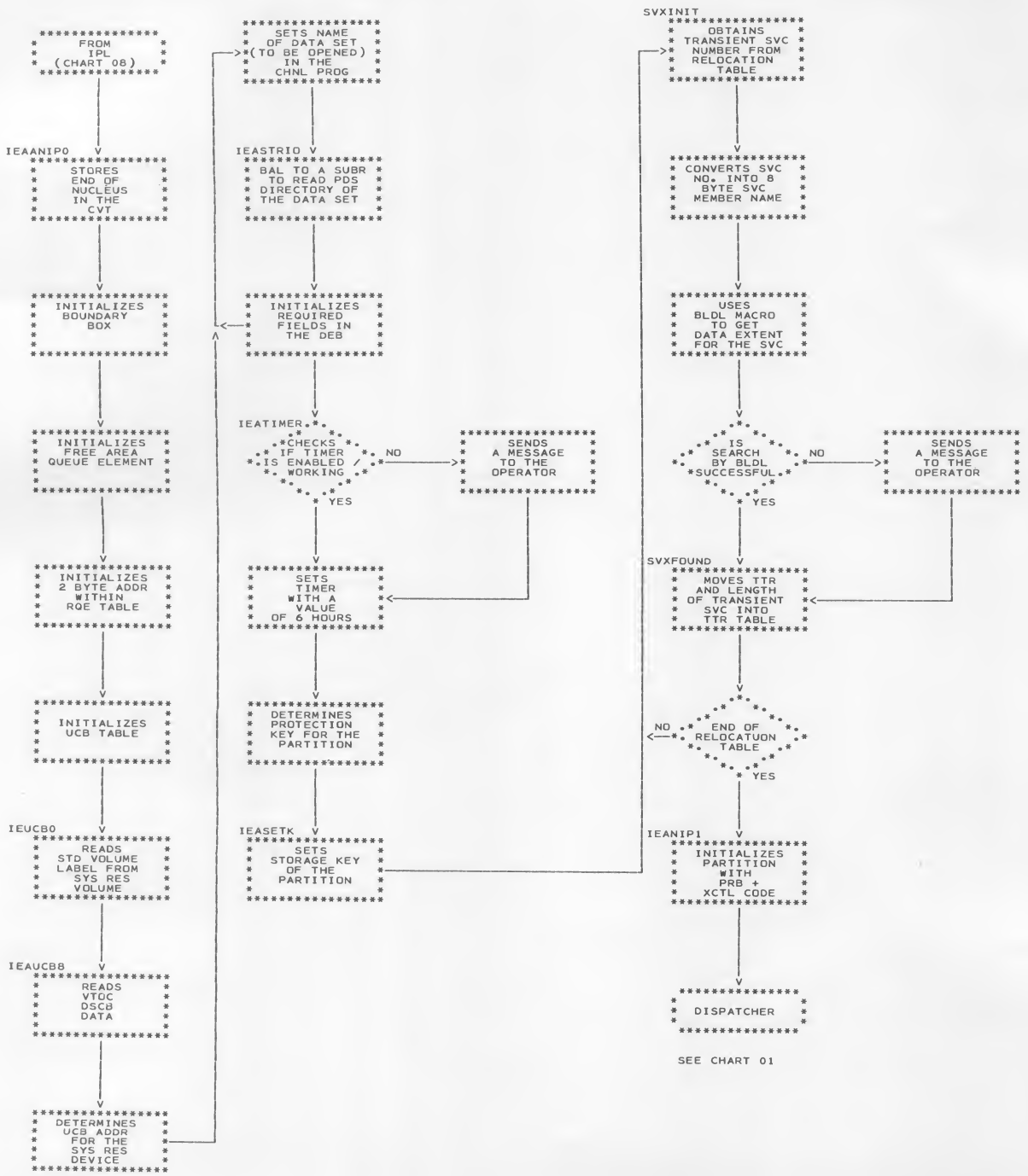


Chart 09. Nucleus Initialization Program Control Flow
(Described in Appendix B)



The initial program loader (IPL) is a service routine that loads into main storage the nucleus and the nucleus initialization program (NIP -- described in Appendix B). IPL is initiated by the operator when he presses the LOAD key on the system control panel. The hardware loads IPL into main storage, IPL loads the nucleus and NIP. On completion, IPL branches to an LPSW instruction in the nucleus, which gives control to NIP.

IPL performs the following major functions:

- Clears main storage and machine registers to correct parity.
- Sets the storage key of main storage to the supervisor protection key, in systems with the protection feature.
- Locates the nucleus on the system residence device.
- Loads the nucleus and NIP.
- Gives control to NIP.

HOW IPL IS ORGANIZED

IPL is made up of two records and eight subroutines:

- IPL Control Record -- This 24-byte record, consisting of an IPL-PSW and two IPL-CCWs, is loaded into main storage at location zero by the hardware circuitry when the operator presses the LOAD key. This record and the IPL bootstrap record are located at track zero, cylinder zero of the system residence device; the IPL subroutines are contained in one record elsewhere on the system residence device.
- IPL Bootstrap Record -- This record, consisting of a chain of CCWs, is loaded into main storage at a location specified by the IPL control record. The IPL bootstrap record loads the IPL subroutines into main storage at location zero.
- Nucleus Selection (IEACOMPR) -- This subroutine selects the nucleus to be loaded.
- Hardware Initialization (IEAMAIN) -- This subroutine clears main storage, machine registers and, where applicable, initializes the storage keys.
- Nucleus Location (IEACOMLP) -- This subroutine locates the nucleus on the system residence device.
- Control Section Data Organization (IEAHOOP) -- This subroutine computes and sequentially arranges nucleus control section data so the nucleus can be loaded into main storage.
- IPL Relocation (IEAADDR) -- This subroutine moves the unexecuted part of IPL to the upper end of main storage to make room for the nucleus.
- Nucleus Load (IEALOAD) -- This subroutine loads the nucleus and NIP into main storage.
- RLD Relocation (IEARELOC) -- This subroutine relocates RLD items within the nucleus text read into main storage.
- Common I/O (IEASTRIO) -- This subroutine, used by IEACOMLP and IEALOAD, issues and tests for the successful completion of START I/O operations.

IPL CONTROL INFORMATION

NIP and the nucleus are combined into one load module and written on the system residence device by the linkage editor at system generation time. IPL is supplied with the fixed name of this "nucleus" load module, but not with its location or the location of its DSCB within the VTOC.

The structure of the nucleus load module on the system residence device is the standard structure described in the publication IBM System/360 Operating System: Linkage Editor, Program Logic Manual. That is, its records and text are ordered as follows:

- Composite ESD Record (CESD).
- Scatter/Translation Record.
- Control Record.
- Text Record (TXT).

- Control/RLD Record (here and elsewhere, RLD data on this type of record depends on the presence of RLD items in the previous text).
- TXT.
- Control/RLD Record.
- TXT.
- and so on, until the end of the load module.

The scatter/translation record is made up of the translation table and the scatter table. The translation table corresponds, entry for entry, to the CESD, where each entry represents one control section (CSECT) made up of a control (or control/RLD) record and TXT. Entry 0 of both the translation table and the scatter table is a dummy entry containing zeros. Entry 1, corresponding to an ESDID of 1, represents NIP, which is the first CSECT of the nucleus load module. The translation table contains 2-byte pointers to the 4-byte entries in the scatter table.

IPL TABLES

Since the order of nucleus CSECTs on the system residence device is not fixed until system generation time, IPL organizes the information available for the CSECTs before loading the text within CSECTs into main storage. IPL organizes the data by creating three tables:

- SIZTABLE -- a table of CSECT sizes.
- ADRTABLE -- a table of addresses where the CSECTs are to be loaded.
- RLFTABLE -- a table of relocation factors.

These tables are arranged in the same sequence as the CSECT entries in the scatter table and have 4-byte entries, making each table the same length as the scatter table.

To make up the SIZTABLE, IPL performs the following:

- Indexes the scatter table by the contents of the translation table entry to determine the address of the scatter table entry corresponding to a CSECT.
- Loads in a register the assembled origin "0" of the CSECT from the scatter table entry.
- Loads in another register the assembled origin "01" of the next CSECT from the consecutive entry in the scatter table.
- Computes the size of the CSECT by subtracting origin "0" from origin "01."
- Stores the size in SIZTABLE in a position relative to the CSECT position in the scatter table.

The size of the CSECT whose linkage-editor assigned origin is available in the last 4-byte entry of the scatter table is computed by subtracting origin "0" from the size of the nucleus which is available in the PDS directory and stored by IPL in the first word of the SIZTABLE which IPL builds behind the scatter table.

To make up the ADRTABLE, IPL performs the following:

- Stores the address where the second CSECT is to be loaded (assumed to be location 0) in the same position in the ADRTABLE as the CSECT occupies in the scatter table.
- Computes the address for the third CSECT by adding the size of the second CSECT to the address of the second CSECT.
- Stores the address for the third CSECT in the same position in the ADRTABLE as the CSECT occupies in the scatter table.
- Repeats the second and third steps above for each ordered CSECT. (Ordered CSECTs are those which must be loaded first and in the order in which they appear in the translation table.)
- Stores the addresses for non-ordered CSECTs, after computing them as they are encountered sequentially following the last of the ordered CSECTs.

The RLFTABLE is similar in structure to the SIZTABLE and ADRTABLE. Its entries are computed by subtracting the linkage-editor assigned origin from the address at which the CSECT is to be loaded.

IPL CONTROL FLOW

As shown in Chart 08, IPL begins with several operator actions and prior conditions (see the publication IBM System/360 Operating System: Operator's Guide, Form C28-6540). The operator selects the system residence device with the

LOAD-UNIT switches and presses the LOAD key. The hardware circuitry resets the CPU, locates track 0, cylinder 0, and loads the IPL control record into location 0. The control record loads the IPL bootstrap record, which, in turn, loads IPL and passes control to the first subroutine via an LPSW instruction. IPL is executed disabled for all interruptions except program interruptions.

IPL clears storage and registers, selects the nucleus or allows the operator to select a non-standard nucleus, sets storage keys where applicable, searches the VTOC and locates the data set containing the nucleus load module. IPL loads the translation table and the scatter table into main storage, relocates part of IPL (if necessary), calculates relocation constants, and loads the nucleus load module. IPL passes control to NIP by branching to an LPSW instruction in the nucleus.

NUCLEUS SELECTION

This subroutine (IEACOMPR) selects the nucleus for loading or allows the operator to choose a different nucleus, by using the ADDRESS-COMPARE switch and the DATA switch. The procedure for operator-selection of the nucleus is given in the publication IBM System/360 Operating System: Operator's Guide.

HARDWARE INITIALIZATION

This subroutine (IEAMAIN) sets correct parity in the:

- General registers.
- Floating point registers, if present.
- Main storage beyond IPL.

In addition, IEAMAIN sets storage keys to the supervisor protection key.

Program interruptions will occur while setting storage keys in machines without the protection feature, or while correcting parity in machines without floating point registers or without maximum main storage capacity. These interruptions are automatically handled by IEAMAIN. Further program interruptions are unexpected, and this subroutine places the machine in a wait state if they occur.

NUCLEUS LOCATION

This subroutine (IEACOMLP) searches for the location of the specified nucleus name on the system residence device and positions the read head of the system residence device at the first text record of the nucleus. IEACOMLP takes the following steps to lcoate the nucleus:

- Picks up the system residence device address stored at location 2 by the hardware circuitry.
- Reads the standard volume label to find the VTOC DSCB address.
- Reads the VTOC DSCB data to determine the number of tracks per cylinder on the system residence device.
- Searches the VTOC to find the DSCB for the partitioned data set (PDS) name.
- Seeks the track where the PDS directory starts.
- Searches the directory for a record containing the name of the nucleus, using the SEARCH EQUAL HIGH KEY command.
- Reads the PDS directory record.
- Determines the address of the scatter translation record on the system residence device from the PDS directory record.
- Finds the scatter translation record and reads it into main storage above IPL.

The nucleus location subroutine uses the common I/O subroutine, IEASTRIO, when reading the standard volume label, VTOC, etc., from the system residence device into main storage. Before using the common I/O subroutine, IEACOMLP sets up a channel program with an appropriate chain of CCWs to SEEK, SEARCH, TIC and READ.

CONTROL SECTION DATA ORGANIZATION

This subroutine (IEAHOOP) computes the address for loading the ordered CSECTs and also computes the relocation factor and size of each CSECT. This data is arranged in tables -- SIZTABLE, ADRTABLE, and RLFTABLE -- for use by the nucleus load subroutine. The tables and the procedures IEAHOOP uses to make them are described under the earlier heading, "IPL Tables."

IPL RELOCATION

This subroutine (IEAADDR) relocates that part of IPL not executed at the time of the loading of the nucleus into the numerically-lower end of main storage. The tables created at the top of IPL are included in the relocation. Space for the RLD information concerning the nucleus is assigned from the top of NIP to the bottom of the relocated portion of IPL.

NUCLEUS LOAD

This subroutine (IEALOAD) loads the nucleus into main storage, placing the relocatable modules into main storage in the order of their position in the translation table. Unless INSERT cards are used for each nucleus CSECT prepared by linkage editor, the order of the loading of the relocatable nucleus CSECTS will vary. IPL sets a buffer of 256 bytes in IPL for reading control/RLD records, and performs the following actions:

- Reads a control/RLD record into the buffer and interrogates the record.
- Picks up from the control/RLD record the ESDID of the text record that follows the control/RLD record.
- Determines the address, L, at which the text record of the CSECT is to be read, by adding the relocation factor from the RLFTABLE to the assigned origin of the record.
- Reads the TXT record of the CSECT at address L.
- Adds the number of text bytes read, T, to address, L, to compute the address where the next text record of the same CSECT is to be read. Sets L = L + T.
- Reads into the buffer the control/RLD record following the text record.
- Builds a table of RLDs by moving RLD information bytes from the control/RLD record and keeps a count of the RLD bytes moved into the RLD table above NIP.
- Repeats the above steps until all the records of the nucleus are read into main storage.

The nucleus load subroutine uses the common I/O subroutine when reading the CCW, control/RLD and TXT records of the nucleus load module from the system residence device into main storage. Before using the common I/O subroutine, IEALOAD sets up a channel program with an appropriate CCW to READ the particular record.

RLD RELOCATION

This subroutine (IEARELOC) scans the RLD table created by IEALOAD and relocates the load constants in the nucleus text, using relocation factors stored by IPL in the RLFTABLE. At the completion of IEARELOC, IPL's work is done and control is passed to NIP.

COMMON I/O

This subroutine (IEASTRIO), used by nucleus locate and nucleus load, issues and tests for the successful completion of START I/O operations. Nucleus locate and nucleus load set up the CAW and CCWs and then branch and link to IEASTRIO. After execution of IEASTRIO, control is returned to the IPL subroutine that branched to it.

Error conditions encountered during the execution of IEASTRIO are indicated to the operator by the WAIT light, and the error type is stored in the address field of the WAIT PSW.

The operator can retry IPL when the WAIT light is on. If IPL is unsuccessful after a few trials, the operator displays the address field of the PSW to determine the error type, and informs the customer engineer. The ten error types are shown in Figure 20.

Error Type	Bit Pattern Displayed	Meaning
1	00000001	I/O is not operational.
2	00000010	I/O operation is not initiated. CSW is stored. Unit is not busy.
3	00000011	I/O operation is not initiated. CSW is not stored. Channel is not busy.
4	00000100	During TEST I/O. Channel is not busy. CSW is not stored.
5	00000101	During TEST I/O. Unit check condition is indicated.
6	00000110	During TEST I/O. Any of these conditions are indicated: Interface control check. Channel control check. Channel Data check. Channel chaining check. Program Check.
7	00000111	During TEST I/O. Unit check condition. Record not found during nucleus location.
8	00001000	Available space for reading RLD records has been exceeded.
9	00001001	Unexpected program interruption. IPL contaminated.
FF	000000FF	No IPL on this direct-access device.

Figure 20. IPL Error Types

APPENDIX B: NUCLEUS INITIALIZATION PROGRAM (NIP)

The nucleus initialization program (NIP) consists of several subroutines, each of which performs an initialization function required by the resident portion (nucleus) of the primary control program. Such functions include opening of the SVC and Link libraries, setting the protection key of main storage and placing the addresses of the upper and lower boundaries of the partition into the boundary box.

The NIP sub-routines are packaged in one non-resident module. The NIP module is processed by the linkage editor together with the nucleus modules. It is loaded into main storage immediately following the nucleus, by the IPL program. NIP is entered from the IPL program and, on completion, passes control to the dispatcher, after which it is overlaid by the processing programs.

NIP operates partially under its own stand-alone input/output routine and partially under system routines including the I/O supervisor. NIP has its own TCB, RB and boundary box, all of which are preassembled within NIP code. The location NEW contains the address of the TCB.

The NIP module initializes the following:

- Communication Vector Table (CVT).
- Partition.
- Boundary Box.
- Free Area Queue Element.
- UCB Table and Request Element Table.
- SYS1.SVCLIB, SYS1.LINKLIB, SYS1.LOGREC DEB.
- SVC Table Extension (optional).
- Protection Key (optional).
- Timer (optional).

NIP CONTROL FLOW

When entered from the IPL program, NIP saves the address of the system residence device, stored in register 10 by the IPL program. (See Chart 09.) It rounds up the address of the end of nucleus to a 2048-byte boundary and stores this value in the CVT for use by the system environment recorder (SER 0).

NIP examines the parameters provided by the user in the boundary box, and determines the addresses of the free area queue element and lower and upper boundaries for the partition. It stores these addresses in the boundary box. It also stores the number of free bytes in the partition in the free area queue element.

NIP changes the 2-byte displacements of each "Forward Link" from the start of the I/O supervisor into absolute addresses and stores them into the request element table. It also changes the 2-byte displacements of each UCB from the start of the I/O supervisor into absolute addresses and stores them into the UCB table.

It initializes the pre-assembled DEB-for SYS1.SVCLIB, SYS1.LINKLIB, and SYS1.LOGREC data set.

NIP optionally determines if the timer is enabled and working. If the timer is not enabled and working, NIP sends a timer-status message to the operator. If the timer is enabled and working, it sets the timer with a value of six hours.

NIP optionally determines the protection key for the partition from the "protect key" field within the TCB associated with the partition. It sets the storage key of the partition.

NIP optionally extends the SVC table to contain the TTR and the length of each transient SVC routine.

It moves a PRB and the XCTL code from the NIP module to the beginning of the partition and relocates the address constants within the XCTL code.

The PRB and XCTL code have been preassembled within the NIP module. The code moved into the partition passes control to the job scheduler through an XCTL macro-instruction.

After completing all the initialization procedures, NIP passes control to the dispatcher.

CVT INITIALIZATION

NIP rounds up the address of the end of nucleus to a double-word boundary and stores it in entry 33 in the CVT. This information is used by system environment recorder.

PARTITION INITIALIZATION

The main storage area outside the fixed area is called the partition.

The boundary between the fixed area and the partition is on a double word for a system without the protection feature.

The initialization of the partition consists of:

- Moving pre-assembled code from NIP to the beginning of the partition. This code includes a PRB and the XCTL code that causes loading of the job scheduler through an XCTL macro-instruction.
- Relocating the address constants in the PRB and XCTL code.

NIP may be overlaid when the pre-assembled code is moved to the beginning of the partition. To eliminate this possibility, NIP code is assembled 2048 bytes from the beginning of the NIP module.

Main storage before and after initializing the partition is shown in Figure 21.

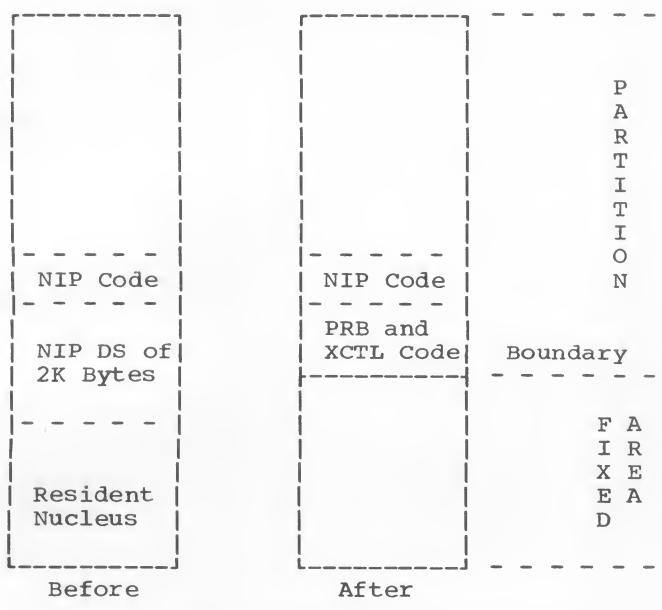


Figure 21. Main Storage Initialization

BOUNDARY BOX INITIALIZATION

A 12-byte boundary box specifies the boundary of the partition. The *parameters* specified by the customer are assembled in the boundary box at System Generation time (Figure 22).

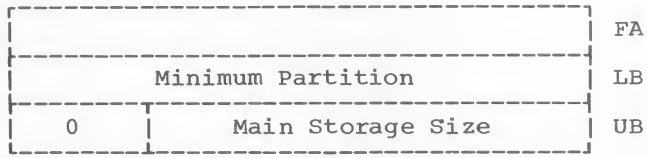


Figure 22. Boundary Box

The initialization of the boundary box consists of computing and storing the following addresses into the boundary box:

Upper boundary of the partition - UB

Lower boundary of the partition - LB

Free area queue element - FA

The boundary box initialization for the single-task supervisor is shown in Figure 23.

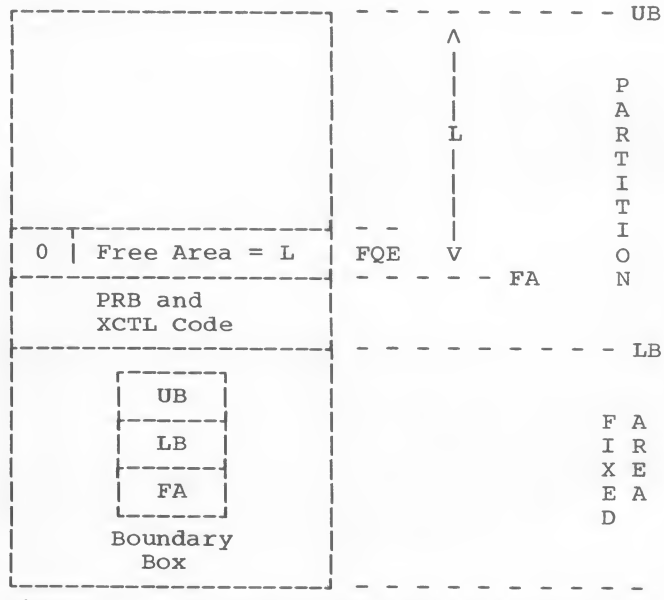


Figure 23. Boundary Box Initialization

The boundary box addresses are computed as follows:

UB = *highest* address in the main storage, computed dynamically.

LB = address of the end of nucleus rounded up to a double word boundary for a system without protection feature.

or address of the end of nucleus rounded up to 2048-byte boundary for a protected system.

FA = address of free area queue element (FQE), described in the following section.

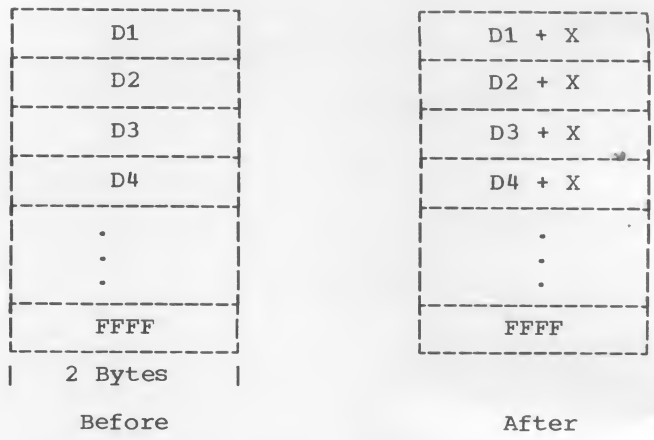


Figure 24. UCB Table Initialization

FREE AREA QUEUE ELEMENT INITIALIZATION

The free area queue element (FQE) is a double word after the PRB and XCTL code within the partition. The initialization of the FQE consists of the following:

- Computing the length of the free area (L) and storing this value in the right half of the FQE. The free area is defined as the partition minus the area occupied by PRB and XCTL Code (see Figure 23).
- Storing zeros in the left half of the FQE.

UCB TABLE AND REQUEST ELEMENT TABLE INITIALIZATION

UCB Table: NIP changes the 2 byte displacements within the UCB table and request element table into 2-byte absolute addresses after these tables are loaded into main storage.

When loaded into main storage, the UCB table contains the displacement (D) of each UCB from the start (X) of the I/O supervisor (IECIOS00). NIP adds X to D and stores the sum into the UCB table.

The start of the UCB table is available in entry 11 within the CVT. The end of the UCB table is indicated by 'FFFF' in the last entry within the UCB table.

Figure 24 shows the UCB table before and after the initialization.

Request Element Table: The request element table consists of a number of request elements (I/O supervisor's name for IQEs with 2-byte link fields; see Chapter 1) that are used to represent I/O interruption requests. The number of elements in the table is determined at system generation and remains fixed.

When loaded into main storage, the request element table contains the displacement (L) of each 'Forward Link' from the start of the I/O supervisor (X). NIP adds X to L and stores the sum into the request element table.

The start of the request element table is available in entry 31 within the CVT. The end of the request element table is indicated by 'FFFF' in the first two bytes of the last entry in the request element table. Figure 25 shows the request element table before and after the initialization.

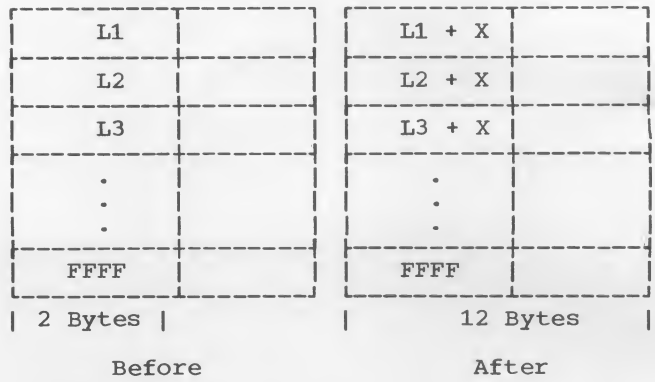


Figure 25. Request Element Table Initialization

In addition to the initialization procedures described above, NIP stores the address of the request element table at the next request element address, which is available in entry 32 in the CVT.

SYS1.SVCLIB, SYS1.LINKLIB, AND SYS1.LOGREC DEB INITIALIZATION

Although the DEB's for these data sets are assembled within the nucleus, some of the DEB fields are not pre-assembled. The data in these fields is stored by NIP to simulate the OPEN function.

The initialization of the DEB consists of determining the following data and storing them into the corresponding fields in the DEB:

- Start cylinder address and track address (CCHH) of the data set.
- End CCHH of the data set.
- Number of tracks occupied by the data set.
- UCB address for the system residence device.
- Appendage address.

Figure 26 shows the DEB fields which are initialized by NIP.

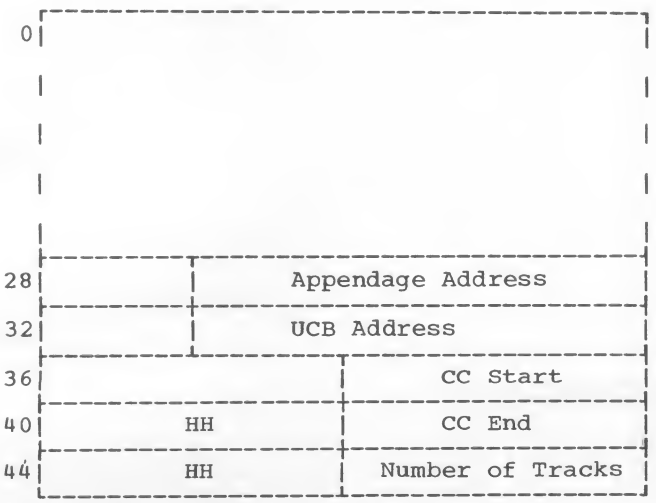


Figure 26. DEB Initialization

NIP executes in a stand-alone environment using its own input/output routine and performs the following functions to accomplish the DEB initialization:

1. Reads the standard volume label to determine the volume table of contents (VTOC) address on the system residence device.
2. Reads the data portion of VTOC DSCB to determine the tracks per cylinder for the system residence device.
3. Determines the UCB address of the system residence device through UCB table look up.
4. Determines the DEB address through the use of CVT and DCB. The DEB Address is available within the corresponding DCB. The DCB addresses for SYS1.SVCLIB, SYS1.LINKLIB, and SYS1.LOGREC data sets are available in the CVT at entries 22, 3, and 30, respectively.
5. Searches the VTOC and reads, into a buffer, the data portion of the DSCB for the data set.
6. Moves Start CCHH and End CCHH for the data set from the buffer into the DEB.
7. Computes the number of tracks contained within the data set extent and stores this value in the DEB.
8. Stores the UCB address into the DEB.
9. Moves the I/O appendage address from entry 6 of the CVT into the DEB.
10. Repeats steps 4-9 for each data set.

SVC TABLE EXTENSION (TTR TABLE) INITIALIZATION

This is an optional NIP function that is selected at system generation time.

The TTR address and length (L) of each non-resident SVC routine are available in the partitioned data set (PDS) directory of the SVC library.

The TTR table initialization consists of the following:

- Searching the PDS directory of the SVC library to find the TTR and length of each transient SVC routine.
- Storing TTR and L of each transient SVC routine in a table within the nucleus. The assigned area for this table is within the SVC handler routine.

The TTR table consists of a 4-byte entry for each transient SVC routine. The format

of each 4-byte entry in the table, is shown in the diagram below:

where

TT = Track address of the transient SVC routine relative to the start of the SYS1.SVCLIB data set.

R = Record number on the track.

L = Length in bytes of the transient SVC routine.

ESA = Extended save area in double words. This field is pre-assembled in the table.

NIP uses the following information available in the SVC handler routine to accomplish the initialization function:

- Relocation table containing 1-byte index number for each SVC in the SVC table.
- Highest number assigned to an SVC provided by IBM.
- Highest number assigned to a resident SVC.

To initialize the TTR table, NIP follows the procedure described below:

1. Constructs an eight byte name for the transient SVC by using the relocation table and the highest resident SVC number, as explained below:
   - Picks up the entry in the relocation table which corresponds to a transient SVC.
   - Translates the entry number in the relocation table to a SVC number.
   - Converts the SVC number from binary to decimal.
   - Unpacks the decimal number to a 4-byte number.
   - Constructs an 8-byte name for the SVC routine by placing the 4-byte unpacked decimal number beside a pre-assembled four character prefix for the SVC names, as follows:

IGC0	XXXX
pre-assembled prefix	unpacked decimal number

2. Loads the following registers:
   - Address of the input parameter list to the BLDL macro-instruction is placed in register 0.
   - Address of the SYS1.SVCLIB DCB is placed in register 1.
3. Issues the BLDL macro-instruction to search the SYS1.SVCLIB directory.
4. Tests for the successful execution of the BLDL macro-instruction.
5. On successful completion, BLDL returns the data extent for the SVC routine in a return area. NIP moves the TTR and length of the SVC routine from the return area into the TTR table, in a format shown in the diagram above.
6. When unsuccessful, BLDL returns an error code in register 15. NIP tests the error code and sends one of the following error messages to the operator:

   "IEA101I SVC ROUTINE IGC0XXXX NOT AVAILABLE - PERMANENT I/O ERROR ON SVC LIBRARY."

   "IEA102I SVC ROUTINE IGC0XXXX NOT AVAILABLE - NOT FOUND ON SVC LIBRARY."
7. Scans the relocation table and repeats the above procedure for each transient SVC routine.

PROTECTION KEY INITIALIZATION

Main storage protection is an optional hardware feature. When protection is selected, the storage keys are set according to the following criteria:

- The storage occupied by the nucleus has a storage key of zero.
- The partition has a non-zero storage key, specified in the TCB associated with the partition.

NIP obtains the storage key for the partition from the "protect key" field within the TCB corresponding to the partition, and sets the partition to the appropriate storage key.

TIMER INITIALIZATION

The timer is an optional hardware feature. It can be enabled or disabled by a switch on the system control panel.

The timer initialization consists of the following:

- Testing to check if the timer is enabled and working.
- Setting the timer to six hours when control is given to the job scheduler.

NIP performs the following functions to initialize the Timer:

1. Tests to check if the timer is working:
   - Sets location 80 with value of six hours (X'6309109E).
   - Waits for the timer to decrement.
   - Compares the contents of location 80 with the original value of six hours. If the contents of location 80 are equal to six hours, sends the following message to the operator:

     "IEA100A TIMER IS NOT WORKING. PUT TIMER SWITCH ON."
2. Resets location 80 with the 6-hour value.

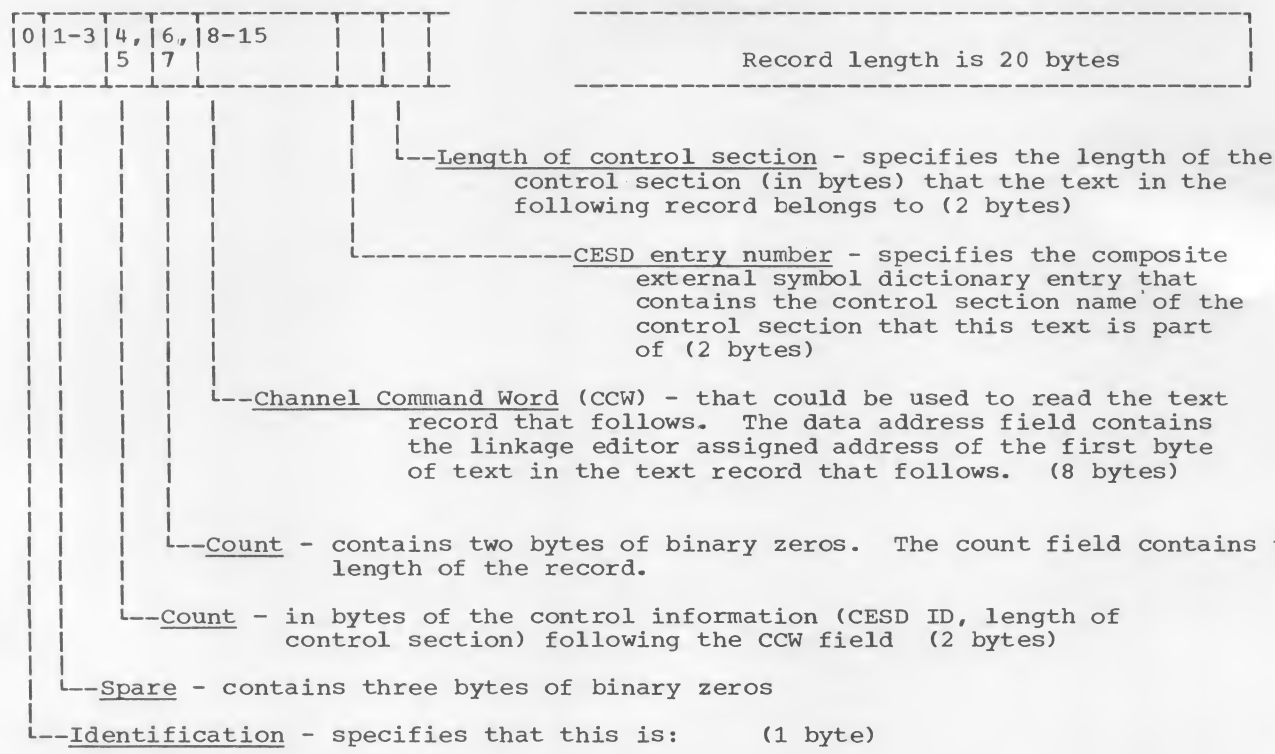
APPENDIX C: GUIDE TO THE LINKAGE EDITOR MAP OF THE NUCLEUS

Csect Names in Order of Appearance in Nucleus	Sysgen Output Macro. to be Checked for Module Name	Microfiche Module Name	Routine Name (or Other Specified Function; e.g., Table)
IEAAIH00	IEAAIH IEAAPS IECIOS	* * *	First Level Interruption Handlers (FLIHs) Dispatcher and Exit Effector I/O Supervisor
IGC009	---	IEAADL00	Delete
IGC012	---	IEAASY00	Synch
IGC010	---	IEAAMS00	Getmain
IEA0PL00	---	IEAAPL00	Prolog
IGC011	---	IEA0RT10	Timer SVC
IEEBA1	---	IEECIR01	Console Interruption (Job Management)
IEA0AB00	---	IEAAAB00	Abterm
IGC001	IEAAWT	*	Wait
IHASVC00	SGIEA2SV	*	SVC Table
IEAATA00	IEAATA	*	Second Level Interruption Handler and SVC 3
IEACVT	CVT	*	Communications Vector Table
IGC002	IEAAPT	*	Post
IGC006	IEAATC	*	Link
IEATCB00	IEATCB	*	Task Control Block
IEWFTMIN	---	IEWFTMIN	Program Fetch
IEFJOB	---	IEFKRESA	Master Scheduler Tables and Work Area (Job Management)
IFBDCB00	---	IFBDCB00	System Environment Recorder (SER) Data Control Block
IGC018	---	IECPFIN1	Find (Data Management)
IGC037	---	IEWSVOVR	Overlay Supervisor
IEEBCIPE	---	IEEBCIPE	External Interruption (Job Management)
IEC2311A	---	IEC2311A	Disk Error Routine (I/O Supervisor)
IEFDPOST	---	IEFDPOST	Unsolicited Interruption (Job Management)
IEEMSLT	SGIEE001	*	Master Scheduler Resident Control Data Area (Job Management)
IECZDTAB	SGIEC0DT	*	Direct Access Device Table (I/O Supervisor)
IECINTRP	---	IECINTRP	Sense and Status Interpreter (I/O Supervisor)
IEAANIP0	IEAANIP	*	Nucleus Initialization Program
*Variable module names, dependent on macro-instruction's use.			

Listing Name	Chart	Routine Name	Entry Point
IEAAWT	02	WAIT	IGC001
IEAAXR00	02	EXTRACT	IGC040
IEABDL00	04	DELETE	same as for IEAADL00
IEABMS00	03	GETMAIN/FREEMAIN	same as for IEAAMS
IEACMS00	03	GETMAIN/FREEMAIN	same as for IEAAMS
IEADMS00	03	GETMAIN/FREEMAIN	same as for IEAAMS
IEAIPL	08	IPL	IEAIPL
IEAQRT00	07	TIME	IGC011
IEAQST00	07	STIMER	IGC047
IEAQTI00	07	TIMER SLIH	IEAQTI00
IEAQTT00	07	TTIMER	IGC046
IEWFTMIN	05	PROGRAM FETCH	IEWMSEPT (from FINCH) IEWFBOSV (from OVERLAY SUPERVISOR)
IEWSVOVR	06	OVERLAY SUPERVISOR 1	IGC037 IGC045
IEWSXOVR	06	OVERLAY SUPERVISOR 2	IEWSXOVR
IEWSYOVR	06	OVERLAY SUPERVISOR 2	IEWSYOVR

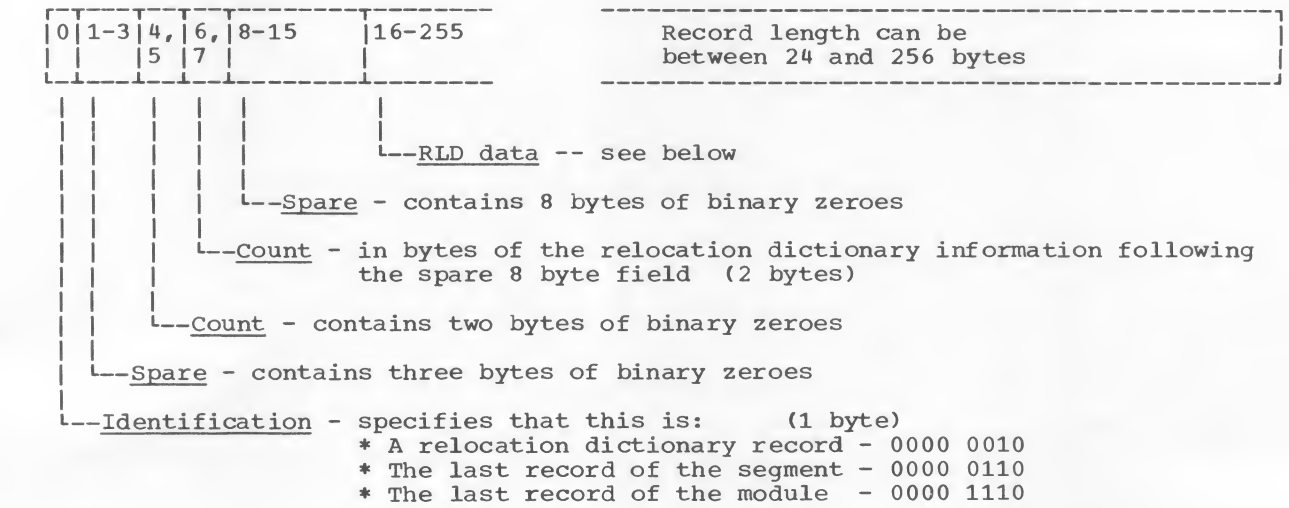
APPENDIX D

CONTROL RECORD - (LOAD MODULE)

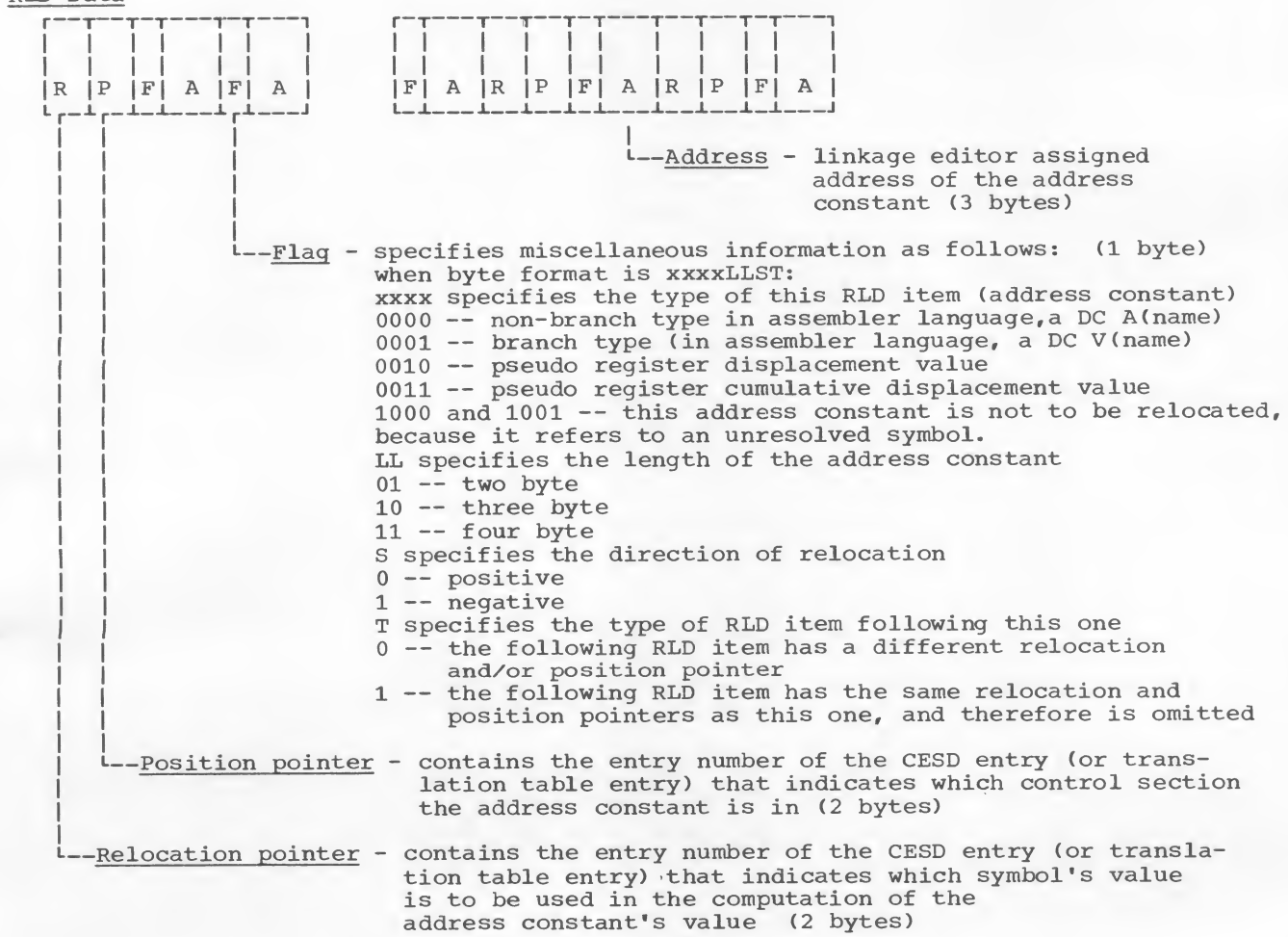


- A control record - 0000 0001
- The control record that precedes the last text record of this overlay segment - 0000 0101
- The control record that precedes the last text record of the module - 0000 1101

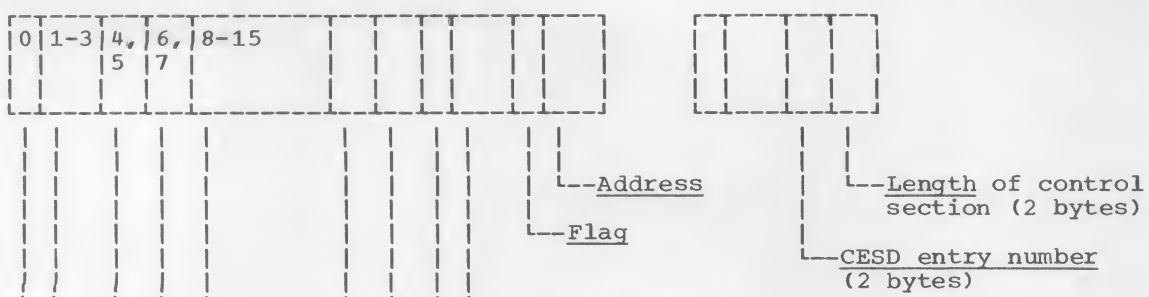
RELOCATION DICTIONARY RECORD - (LOAD MODULE)



RLD Data



CONTROL AND RELOCATION DICTIONARY RECORD - (LOAD MODULE)



- 8-15 — Address
- 6, 7 — Flag
- 4, 5 — Length of control section (2 bytes)
- 1-3 — CESD entry number (2 bytes)

- Address (3 bytes)
- Flag (1 byte)
- Position pointer (2 bytes)
- Relocation pointer (2 bytes)
- Channel Command Word (8 bytes)
- Count of RLD information (2 bytes)
- Count of control information (2 bytes) - the control information contains the ID and length of control sections in the following text record.
- Spare (3 bytes)
- Identification (1 byte) - specifies that this record is:

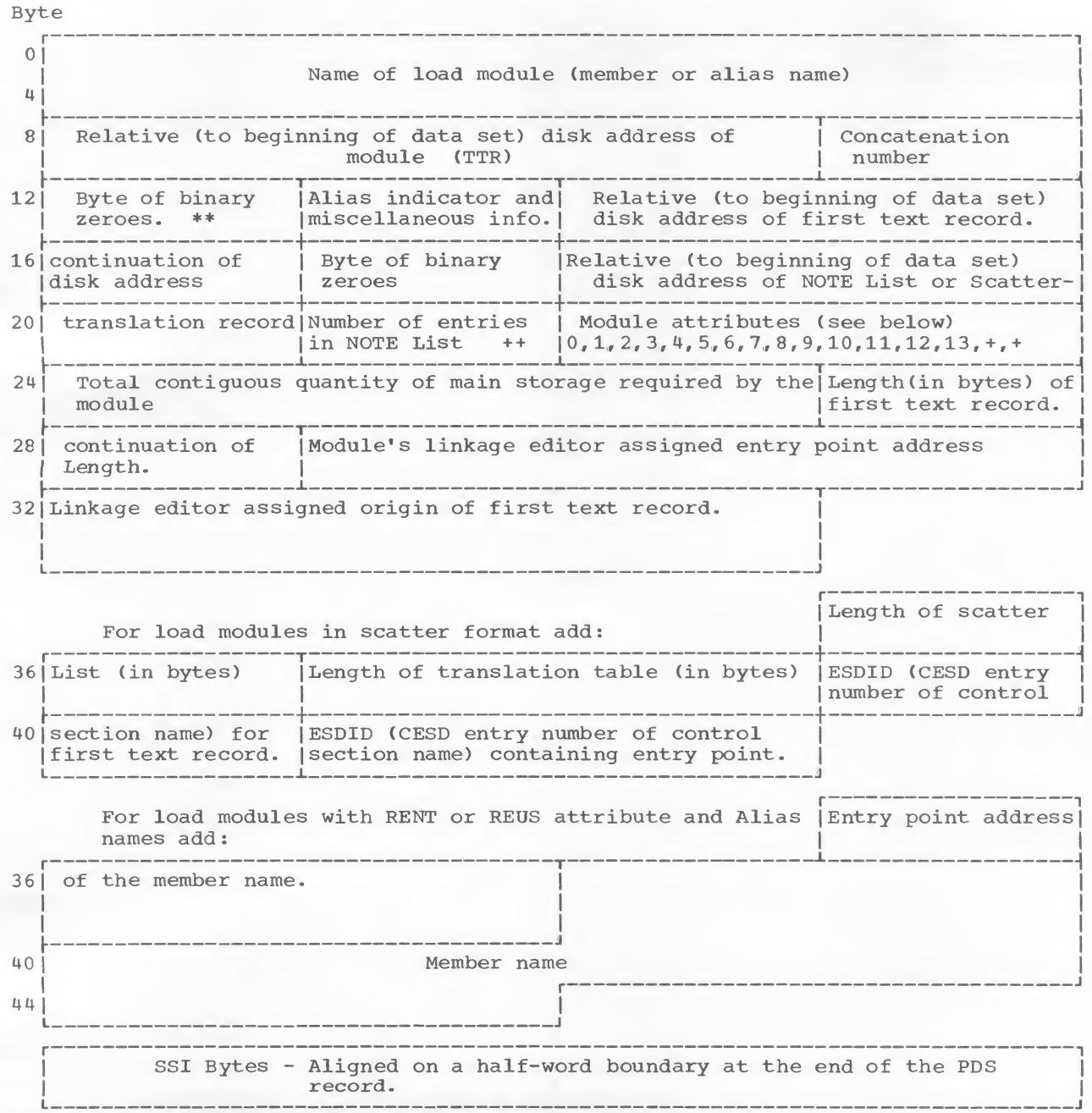
  - A control and RLD record - 0000 0011
  - A control and RLD record that is followed by the last text record of a segment - 0000 0111
  - A control and RLD record that is followed by the last text record of a module - 0000 1111

Note: For detailed descriptions of the data fields see:

Relocation Dictionary Record
Control Record

The record length will vary from 20 to 260 bytes.

PARTITIONED ORGANIZATION DIRECTORY RECORD - (AS RECEIVED FROM BLDL)



Alias indicator and miscellaneous Information:
1. Alias indicator -- 0 signifies none,1 signifies alias -- bit 0
2. Number of relative disk addresses (TTR)in user data field -- bits 1,2
3. Length of user data field (in halfwords) -- bits 3-7

PDS Directory Record size (for SSI, add 4 bytes to sizes):

Block format	36 bytes *	Scatter format	44 bytes
Block format with alias names	46 bytes	Scatter format with alias names	54 bytes

\+ Reserved
++ This byte contains zero if load module is not in overlay

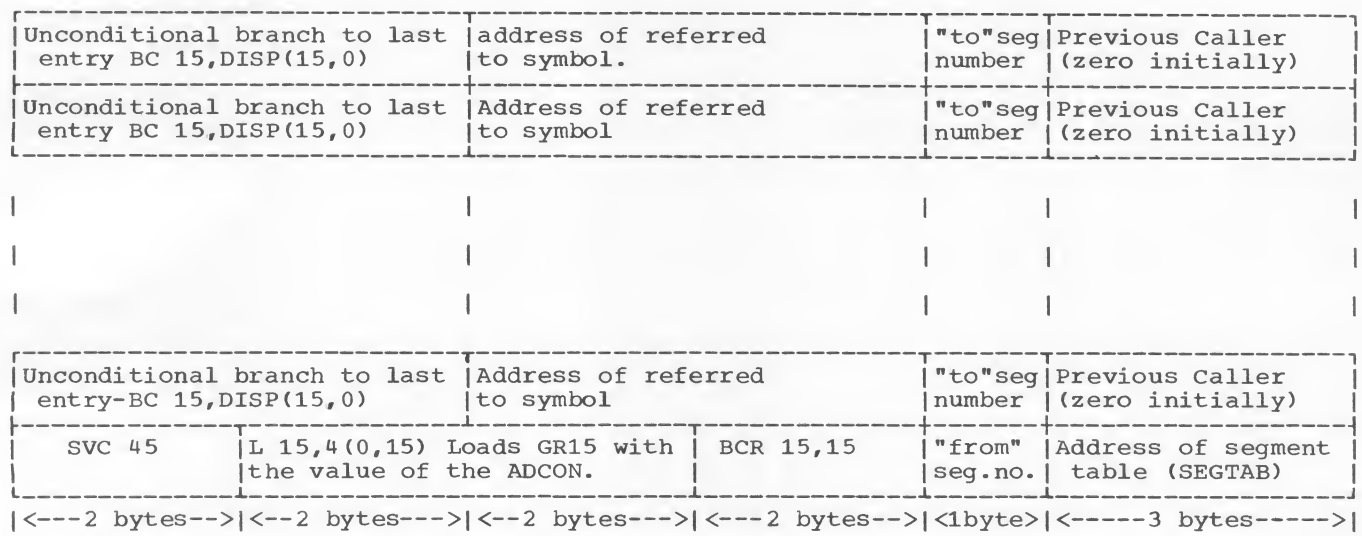
MODULE ATTRIBUTES

Bit Number	Attribute	Bit setting	Indication
0	RENT	0	Not reenterable
		1	Reenterable
1	REUS	0	Not reusable
		1	Reusable
2	OVLY	0	Not an overlay module
		1	Overlay module
3	TEST	0	Not under test
		1	Under test
4	LOAD	0	Not only loadable
		1	Only loadable *
5	Format	0	Block format
		1	Scatter format
6	Executable	0	Not executable
		1	Executable
7	Format	0	Module contains more than one text record and/or RLD record(s).
		1	Module contains only one text record and no RLD record.
8	Compatibility	0	Module can be processed by all levels of linkage editor.
		1	Module cannot be reprocessed by linkage editor-E.
9	Format	0	Linkage editor assigned origin of first text record is not zero.
		1	Linkage editor assigned origin of first text record is zero.
10	Format	0	Linkage editor assigned entry point is not zero.
		1	Linkage editor assigned entry point is zero.
11	Format	0	Module contains RLD record(s)
		1	Module does not contain an RLD record.
12	Editability	0	Module can be reprocessed by linkage editor.
		1	Module cannot be reprocessed by linkage editor.
13	Format	0	Module does not contain TESTRAN symbol records.
		1	Module contains TESTRAN symbol records.
14	Reserved		
15	Reserved		

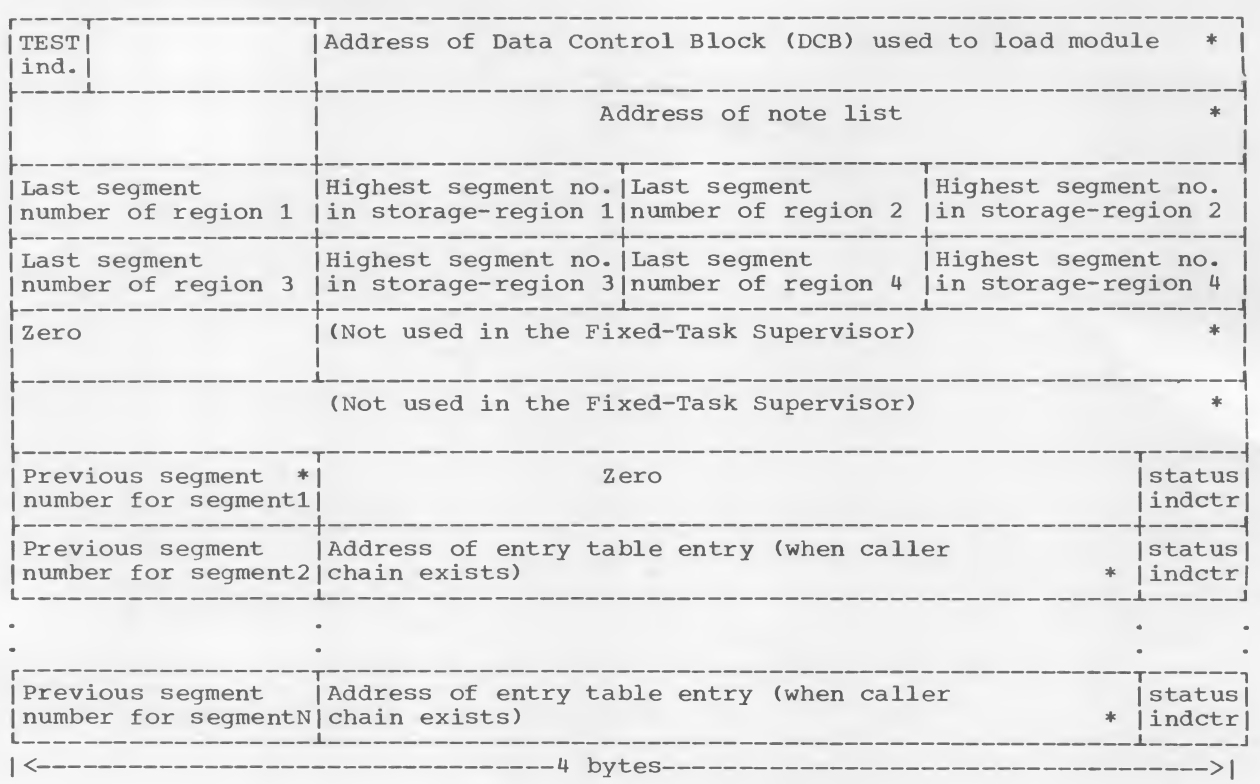
* Module can only be loaded with the LOAD macro-instruction. When the module is in main storage it will be entered directly, and not through the use of an XCTL, LINK, or ATTACH macro-instruction.

** This is normally a zero byte inserted to maintain half-word boundaries. If the DCB operand was specified as zero, this byte will contain a 1 if the name was found in the link library, and a 2 if the name was found in the job library.

ENTRY TABLE (ENTAB)



SEGMENT TABLE (SEGTAB)



TEST indicator -- specifies that this module is "under test" using TESTRAN.
(Bit 1) Initialized by program fetch.

Highest segment no. in storage -- is initially set to 00 except for region 1 which is initially set to 01 by linkage editor.

Status indicator -- indicates the status of this segment with the two last bits of the entry table address field as follows:

00 -- segment is in main storage as a result of a branch to the segment.
10 -- segment is in main storage, no caller chain exists.
01 -- segment is not in main storage, but is scheduled to be loaded.
11 -- segment is not in main storage.

The status indicator for segment 1 is initially set to 10, all the rest are initially set to 11.

* Set to zero by linkage editor.

Form Y28-6612-1

Printed in U.S.A.

Y28-6612-1

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

CUT ALONG LINE

READER'S COMMENTS

Title: IBM System/360 Operating System
Fixed-Task Supervisor
Program Logic Manual

Form:Y28-6612-0

Is the material:	Yes	No
Easy to Read?	___	___
Well organized?	___	___
Complete?	___	___
Well illustrated?	___	___
Accurate?	___	___
Suitable for its intended audience?	___	___

How did you use this publication?
___ As an introduction to the subject ___ For additional knowledge
Other _____

Please check the items that describe your position:
___ Customer personnel ___ Operator ___ Sales Representative
___ IBM personnel ___ Programmer ___ Systems Engineer
___ Manager ___ Customer Engineer ___ Trainee
___ Systems Analyst ___ Instructor Other_____

Please check specific criticism(s), give page number(s),and explain below:
___ Clarification on page(s)
___ Addition on page(s)
___ Deletion on page(s)
___ Error on page(s)

Explanation:

Name_____
Company_____
Address_____
City _____
State _____Zip Code _____

FOLD ON TWO LINES,STAPLE AND MAIL
No Postage Necessary if Mailed in U.S.A.

Form Y28-6612-1

staple

fold fold

FIRST CLASS
PERMIT NO. 81

POUGHKEEPSIE, N.Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
P.O. BOX 390
POUGHKEEPSIE, N. Y. 12602

ATTN: PROGRAMMING SYSTEMS PUBLICATIONS
DEPT. D58

fold fold

Printed in U.S.A.

Y28-6612-1

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

staple